

Szerkesztette:
Nyékyné Gaizler Judit

Az Ada95 programozási nyelv

ELTE EÖTVÖS KIADÓ

Szerzők:

Belokosztolszki András	Az Input-Output használata
	A Függ. (Előre definiált nyelvi környezet)
Czabala Péter	Típusok és kifejezések
Horváth Zoltán	Lexikális elemek
	Utasítások
	A csomag
	Hatáskör és láthatóság
	A program szerkezete
	Taszkok
Kozics Sándor	Az Ada83-mal kapcsolatos részek
	A magasszintű programozási nyelvek alapfogalmai
Kozsik Tamás	Alprogramok
	Kivételek, kivételkezelés
	Sablonok
	Védett típusok és objektumok
Nyékyné Gaizler Judit	Bevezetés
	Az Ada nyelv áttekintése
	Típusok és kifejezések
	Objektum-orientált programozás
	Konténer típusok
	Adatfolyamok
	Az Input-Output használata
	A Függ. (Előre definiált nyelvi környezet)
	A magasszintű programozási nyelvek alapfogalmai
Pécsy Gábor	Reprezentációs specifikációk
	E Függ. (Elosztott rendszerek)
Steingart Ferenc	Az Input-Output használata
	A Függ. (Előre definiált nyelvi környezet)
	B Függ. (Kapcsolódás más nyelvekhez)
	C Függ. (Rendszerprogramozás)
	D Függ. (Valós idejű rendszerek)
	F Függ. (Információs rendszerek)
	G Függ. (Numerikus műveletek)
Zsók Viktória	Típusok és kifejezések
	Taszkok

Lektorok:

Kiss Viktória, Gyaraki György, Kálmán György, Kiss Kata

Az
Ada95
programozási nyelv

EGYETEMI TANKÖNYV

Szerkesztette:
NYÉKYNÉ GAIZLER JUDIT

ELTE Eötvös Kiadó, Budapest

A tankönyv a Művelődési és Közoktatási Minisztérium támogatásával, a Felsőoktatási Pályázatok Irodája által lebonyolított felsőoktatási tankönyvtámogatási program keretében jelent meg.

A Szerzők és a Kiadó e könyv tartalmi és formai összeállítása során a legjobb tudásuk szerint jártak el. Ennek ellenére nem kizárható, hogy a könyvben előfordulnak hibák. A könyv, illetve a benne levő ismeretanyag használatából közvetlen vagy közvetett módon származó károkért sem a Szerzők, sem pedig a Kiadó nem vállal felelősséget.

Ezen kiadvány egészének vagy egy részének a másolása, reprodukálása (bármilyen formában), valamint lefordítása más nyelvekre kizárólag a Kiadó előzetes írásbeli hozzájárulásának birtokában engedélyezett.

A UNIX az X/Open bejegyzett védjegye.
A Java a Sun Microsystems, Inc. védjegye.
A Windows a Microsoft, Corp. védjegye.
A könyvben előforduló más védjegyek nevei nagy kezdőbetűkkel vannak írva, ha a szerzőknek tudomása van az adott védjegyről.

© Belokosztolszki András, Czabala Péter, Horváth Zoltán, Kozics Sándor, Kozsik Tamás, Nyékyné Gaizler Judit, Péchy Gábor, Steingart Ferenc, Zsók Viktória 1998.

ISBN 963 463 238 6

A fedelet Dukay Barna tervezte
Lady Ada Augusta Byron képét a
http://wuarchive.wustl.edu/languages/ada/userdocs/htm/cardcat/a_graphi.htm/
szabadon felhasználható ftp-site-ról töltöttük le. A fájl neve: ladyada.ps

Felelős kiadó: az ELTE Eötvös Kiadó vezetője

Készült a *mondAt Kft.* nyomdájában
Felelős vezető: Nagy László
Tel.: 06–30–944–9332

Tartalomjegyzék

Előszó 13

Bevezetés 15

1. Az Ada nyelv áttekintése 25

1.1. Az Ada kialakulásának történeti háttere . . . 25
1.2. Az Ada rövid jellemzése . . . 27

2. Lexikális elemek 31

2.1. Feladatok . . . 33

3. Típusok és kifejezések 35

3.1. Típusok . . . 35
3.1.1. Altípusok . . . 38
3.1.2. Származtatott típusok . . . 40
3.2. Objektum deklarációk és értékadások . . . 42
3.3. Skalár típusok . . . 45
3.3.1. Attribútumok . . . 46
3.3.2. A diszkrét típusok . . . 48
3.3.3. A valós típusok . . . 56
3.4. A tömb típus . . . 61
3.4.1. Tömb típusú objektumok műveletei . . . 64
3.4.2. Szöveg típusok . . . 67
3.5. A rekord típus . . . 68
3.5.1. A rekord diszkriminánsai . . . 71
3.5.2. A variáns rekord . . . 73
3.6. Mutató típusok . . . 77
3.7. Feladatok . . . 89

4. Utasítások 91

4.1. Egyszerű utasítások . . . 91
4.1.1. Az értékadás . . . 91
4.1.2. A `null` utasítás . . . 92

4.1.3. Alprogramok hívása és a `return` utasítás 92
4.2. Összetett utasítások 93
4.2.1. Elágazás: `if-then-elsif-else, case` 93
4.2.2. Ciklus utasítások: `for, while, exit` 94
4.2.3. A blokk és a `declare` utasítás 96
4.3. A `goto` utasítás . 98
4.4. Feladatok . 98

5. Alprogramok 101
5.1. Paraméterek . 104
5.1.1. A paraméterek módja 105
5.1.2. Másolat és hivatkozás szerinti paraméterátadás . . . 107
5.1.3. Mutató paraméter 110
5.1.4. Az aktuális paraméter megfeleltetése a formálisnak . 112
5.1.5. Paraméterek feltételezett értéke 112
5.2. Alprogram definiálása 113
5.2.1. Megvalósítás megadásának módjai 116
5.2.2. További, az alprogramok közé sorolt entitások 116
5.3. Alprogramok átlapolása 117
5.3.1. Operátorok átlapolása 118
5.4. Alprogramok a könyv további fejezeteiben 120
5.5. Feladatok . 120

6. Kivételek, kivételkezelés 123
6.1. Kivételkezelés az Ada nyelvben 125
6.1.1. A kivételek fajtái 126
6.1.2. A kivételkezelő rész 127
6.1.3. Kivételek kiváltódása, terjedése 130
6.1.4. Kivételek kiváltása, terjesztése 131
6.1.5. Kivétel példányok 132
6.2. Példák kivételek használatára 134
6.2.1. Mire használhatjuk a kivételeket? 134
6.2.2. További példák 137
6.3. Az Ada kivételkezelésének hiányosságai 139
6.4. Utalás a következő fejezetekre 140
6.4.1. Az előre definiált könyvtárak kivételei 140
6.5. Feladatok . 141

7. A csomag 143
7.1. A csomag specifikációja 143
7.2. A csomag törzse . 144
7.3. Az átlátszatlan típus 146
7.3.1. Késleltetett definíciójú konstansok 148
7.3.2. Az átlátszatlan típusok műveletei 148

7.4. A korlátozott átlátszatlan típus . . . 150
7.5. Átlátszatlan kiterjesztések . . . 152
7.6. Illeszkedési szabályok . . . 152
7.7. Példaprogramok . . . 153
7.7.1. Véletlenszámok előállítása . . . 153
7.7.2. Halmaz megvalósítása . . . 154
7.7.3. A Kupac osztály megvalósítása . . . 155
7.8. Ellenőrzött típusok . . . 158
7.9. Gyermekcsomagok . . . 159
7.10. Feladatok . . . 162

8. Hatáskör és láthatóság 163
8.1. Deklarációs környezet . . . 163
8.2. Hatáskör . . . 164
8.3. Láthatóság . . . 165
8.3.1. A `use` utasítás . . . 168
8.4. Az átnevezés . . . 169

9. A program szerkezete 171
9.1. Partíciók, fordítási egységek . . . 171
9.2. Könyvtári elemek . . . 173
9.3. Alegységek . . . 175
9.4. Szemantikus függőség . . . 176
9.5. Példaprogramok . . . 177
9.5.1. Prioritási sorok . . . 177
9.6. Kereszteferencia lista készítése . . . 180
9.7. Feladatok . . . 187

10. Az Input-Output használata 189
10.1. Külső fájlok és fájlobjektumok . . . 189
10.2. Soros és közvetlen elérésű fájlok . . . 190
10.2.1. Szekvenciális fájlok . . . 193
10.2.2. Közvetlen elérésű fájlok . . . 194
10.3. A `Storage_IO` csomag . . . 196
10.4. Textfájlok kezelése . . . 197
10.4.1. Sor- és oldalhosszok beállítása és lekérdezése . . . 201
10.4.2. Oszlopok, sorok, és oldalak műveletei . . . 202
10.4.3. `Get` és `Put` eljárások . . . 206
10.4.4. Egész típusok input-outputja . . . 210
10.4.5. Valós típusok input-outputja . . . 212
10.4.6. Felsorolási típusok input-outputja . . . 215
10.5. `Wide_Character` és `Wide_String` input-output . . . 217
10.6. Adatfolyam input-output . . . 217
10.6.1. A `Streams.Stream_IO` csomag . . . 217

10.7. A Text_IO.Text_Streams csomag 220
10.8. A Wide_Text_IO.Text_Streams csomag 221
10.9. Az input-output során fellépő kivételek 221
10.10. Fájlok megosztása . 222
10.11. Az SVGALib könyvtár . 223
10.11.1. Az svgalib.ads 223
10.11.2. Az svgalibmouse.ads 225
10.12. Feladatok . 226

11. Sablonok **227**
11.1. Sablonok definiálása . 228
11.2. Sablonok példányosítása 231
11.2.1. A sablon-szerződés modell 232
11.3. Sablon paraméterek . 234
11.3.1. Paraméter objektumok 234
11.3.2. A paraméter típusok fajtái 235
11.3.3. Paraméter alprogramok 247
11.3.4. Csomagparaméterek 250
11.4. Példák sablonok használatára 253
11.4.1. Sablonok az előre definiált könyvtárakban 253
11.4.2. Sablonban, illetve sablonként megvalósított típusok . 253
11.4.3. Üres sablon csomag, mint absztrakció 255
11.4.4. Kivételek sablon formális paraméterként 257
11.5. Feladatok . 257

12. Objektum-orientált programozás **259**
12.1. Az objektum-orientált tervezés 259
12.2. Öröklődés . 262
12.3. Primitív műveletek . 266
12.4. Konverzió . 267
12.5. Jelölt típusok a variáns rekord helyett 268
12.6. Többalakúság . 273
12.6.1. Osztályok és futásidejű azonosítás 273
12.6.2. Kiválasztás . 276
12.6.3. Egy lehetséges megvalósítás 278
12.6.4. Az objektumok címkéje és a 'Tag attribútum 278
12.6.5. Egyenlőség és tartalmazás operátorok 278
12.6.6. Eltakart implementáció hívása 280
12.6.7. Újrakiválasztás . 280
12.7. Absztrakt típusok és műveletek 282
12.8. Konstruktorok, destruktorok, értékadás 286
12.9. Feladatok . 287

13. Konténer típusok 289
13.1. A `Lista` típus . 289
13.2. A `Lista` típus megvalósítása 291
13.3. Esemény-listák . 299
13.4. Feladatok . 302

14. Adatfolyamok 303
14.1. Az `Ada.Streams` csomag 303
14.1.1. Adatfolyam-attribútumok 304
14.2. Objektum-hierarchiák tárolása adatfolyamokban 306
14.3. Feladatok . 308

15. Taszkok 309
15.1. A taszk specifikációja és törzse 310
15.2. Taszk típusok, taszk objektumok 312
15.3. Taszkok elindítása, végrehajtása 313
15.4. Taszkok terminálása 313
15.5. A taszkok attribútumai 315
15.6. Belépési pont, belépési pont hívás, `accept` utasítás 315
15.7. Belépésipont-családok 318
15.8. A `select` utasítás 318
15.8.1. A szelektív várakoztatás 319
15.8.2. A `select` utasítás hívó taszkban 323
15.9. Kivételkezelés . 324
15.10. Példák . 324
15.11. Feladatok . 332

16. Védett típusok és objektumok 333
16.1. Áttekintés . 334
16.2. Védett egységek készítése 337
16.2.1. Típus és objektum 338
16.2.2. Szerkezet . 339
16.2.3. Diszkrimináns 340
16.2.4. Őrfeltételek . 341
16.2.5. A műveletek belseje 342
16.2.6. Működés . 344
16.3. A védett egységek használata 347
16.3.1. Értékelés . 349
16.4. Példák . 350
16.4.1. Korlátos buffer 351
16.4.2. Üzenetszórásos szignál 351
16.5. Feladatok . 353

17. Reprezentációs specifikációk 355

17.1. Reprezentációs elemek . . . 355
17.1.1. A `Pack` direktíva . . . 356
17.1.2. Reprezentációs attribútumok . . . 356
17.1.3. Felsorolási típus reprezentációs záradékai . . . 359
17.1.4. Rekordok kialakításának szabályozása . . . 360
17.1.5. Fagyasztási szabályok . . . 362
17.2. A System csomag . . . 363
17.3. Assembly részek beillesztése . . . 365
17.4. Ellenőrzés nélküli adatelérés . . . 366
17.4.1. Ellenőrzés nélküli típuskonverzió . . . 366
17.4.2. Nem ellenőrzött referenciaképzés . . . 368
17.5. Memóriagazdálkodás . . . 369
17.5.1. Szabadlista-kezelés . . . 369
17.5.2. Ellenőrzés nélküli tárfelszabadítás . . . 371
17.5.3. A `Controlled` pragma . . . 372
17.6. További fordítási direktívák . . . 372

A. Előre definiált nyelvi környezet 377

A.1. A `Standard` csomag . . . 377
A.2. Karakterkezelés . . . 383
A.3. Sztringkezelés . . . 386
A.3.1. Fix hosszúságú sztringek kezelése . . . 390
A.3.2. Korlátos hosszúságú sztringek kezelése . . . 397
A.3.3. Nemkorlátos hosszúságú sztringek kezelése . . . 406
A.3.4. Sztringkezelést támogató halmazok és leképezések . . . 413
A.3.5. `Wide_String` kezelés . . . 414
A.4. A numerikus csomagok . . . 414
A.4.1. Elemi függvények . . . 414
A.4.2. Véletlenszámok generálása . . . 418
A.4.3. A lebegőpontos típusok attribútumai . . . 420
A.4.4. A fixpontos típusok attribútumai . . . 424
A.5. Input-Output . . . 425
A.5.1. A `Sequential_IO` sablon csomag deklarációja . . . 425
A.5.2. A `Direct_IO` sablon csomag deklarációja . . . 427
A.5.3. A `Storage_IO` sablon csomag deklarációja . . . 428
A.5.4. A `Text_IO` csomag deklarációja . . . 429
A.5.5. A `Streams.Stream_IO` csomag deklarációja . . . 437
A.5.6. A `Text_IO.Text_Streams` csomag deklarációja . . . 439
A.5.7. A `Wide_Text_IO.Text_Streams` csomag . . . 439
A.5.8. Az `IO_Exceptions` csomag deklarációja . . . 440
A.6. A `Command_Line` csomag . . . 440

B. Kapcsolódás más nyelvekhez 443
B.1. A kapcsolódást leíró pragmák . . . 443
B.2. Az Interfaces csomag . . . 445
B.3. Kapcsolódás a C nyelvhez . . . 446
B.3.1. Az Interfaces.C.Strings csomag . . . 450
B.3.2. Az Interfaces.C.Pointers sablon csomag . . . 452
B.4. Kapcsolódás a COBOL nyelvhez . . . 455
B.5. Kapcsolódás a Fortran nyelvhez . . . 461

C. Rendszerprogramozás 465
C.1. A gépi utasítások elérése . . . 465
C.2. Megszakításkezelés . . . 465
C.3. Védett kezelőeljárások . . . 466
C.4. Az Interrupts csomag . . . 467
C.5. A Discard_Names pragma . . . 468
C.6. Osztott elérésű változók . . . 469
C.7. Taszkok azonosítása . . . 470
C.8. A Task_Attributes sablon csomag . . . 471

D. Valós idejű rendszerek 473
D.1. Taszk prioritások . . . 473
D.2. Prioritás ütemezés . . . 475
D.2.1. A taszk ütemezési modell . . . 475
D.2.2. A szabványos taszk ütemező technika . . . 476
D.3. Maximált prioritás kizárási technika . . . 477
D.4. Belépési pont besorolási technikák . . . 478
D.5. Dinamikus prioritáskezelés . . . 479
D.6. Monoton idő . . . 479
D.7. Szinkron taszk vezérlés . . . 482
D.8. Aszinkron taszk vezérlés . . . 483

E. Osztott rendszerek 485
E.1. Partíciók . . . 485
E.2. A könyvtári egységek osztályozása . . . 486
E.2.1. Osztott passzív könyvtári egységek . . . 487
E.2.2. Távoli típus könyvtári egységek . . . 488
E.2.3. Távoli hívás interfész könyvtári egységek . . . 489
E.3. Távoli alprogramhívások . . . 490
E.4. Dinamikus összekapcsolás . . . 492
E.4.1. Alprogramra vonatkozó mutató típus . . . 492
E.4.2. Osztályszintű korlátozott típusra mutató típus . . . 493

F. Információs rendszerek **497**
F.1. A gépi alapszám definiálása 497
F.2. A `Decimal` csomag 497
F.3. Decimális típusok formázott megjelenítése 498
F.4. A `Text_IO.Editing` csomag 501
F.5. A `Wide_Text_IO.Editing` csomag 504

G. Numerikus műveletek **505**
G.1. Komplex aritmetika 505
G.1.1. Komplex típusok 505
G.1.2. Komplex elemi függvények 510
G.1.3. Komplex input-output 513
G.1.4. A `Wide_Text_IO.Complex_IO` csomag 515
G.2. Hatékonysági követelmények 515
G.2.1. A lebegőpontos aritmetika modellje 516
G.2.2. Lebegőpontos típusok modellorientált attribútumai . 517

H. A magasszintű programozási nyelvek alapfogalmai **519**
H.1. A számítási modellek fajtái 519
H.2. Neumann-elvű nyelvek 520
H.3. A nyelvi elemek, alapfogalmak 520
H.3.1. Általános fogalmak 520
H.3.2. Programszerkezet 521
H.3.3. Deklarációk 521
H.3.4. Memóriakezelés 523
H.3.5. Paraméterátadás 524

I. Feladatok **527**

J. Szótár **553**

Irodalomjegyzék **557**

Tárgymutató **559**

Előszó

Az Ada programozási nyelv tervezése jelentős lépés volt egy új, nagy megbízhatóságú, alap-szoftver készítésére is alkalmas nyelv definiálására. A tervezők figyelembe vették a programozási módszertan elvárásait, s a kialakított nyelvi eszközök számottevően megkönnyítik a nagyméretű, megbízható, újrafelhasználható programok, programrendszerek írását.

Könyvünkkel elsősorban a számítástechnikai és informatikai szakemberek, valamint a felsőfokú tanulmányaik keretében programozási nyelvekkel foglalkozó egyetemi és főiskolai hallgatók munkáját szeretnénk segíteni. Feltételezünk bizonyos mennyiségű informatikai előismereteket olvasóinktól: legalább annyit, hogy ismernek egy szövegszerkesztő programot, mellyel a programok forráskódját a számítógépbe tudják írni.

Köszönetnyilvánítások

A szerzők ezúton köszönik meg Benczúr Andrásnak és Kozma Lászlónak, hogy a könyv megírása során szabadon felhasználhatták, és fel is használták az ELTE TTK Informatika Tanszékcsoportjának infrastruktúráját. Enélkül ez a könyv nem készülhetett volna el.

Megköszönik az ELTE programozó és programtervező matematikus hallgatóinak, valamint a KGRE informatikus hallgatóinak, hogy fordításaikkal, példaprogramjaikkal hozzájárultak az Ada83 és az Ada95 közötti különbségek feldolgozásához, és hatékonyan közreműködtek a sajtóhibák számának csökkentésében, továbbá megköszönik Lakatos Attilának, hogy segített e könyv jelenlegi formájának kialakításában.

Egy ilyen nagy terjedelmű könyvben annak ellenére, hogy a szerzők és a szerkesztők a legjobb tudásuk szerint állították össze, előfordulhatnak hibák. Megkérjük az Olvasót, ha ilyet talál, akkor ezt írja meg elektronikus levélben az *adamegj@ada.inf.elte.hu* címre. A szerzők minden konstruktív kritikát örömmel fogadnak.

Bevezetés

A programozási nyelvekről sokan hiszik, hogy csak jelölésformát adnak a program leírásához, s nem veszik figyelembe – esetleg nem is tudnak róla –, hogy az ún. magasszintű vagy felhasználó orientált nyelvek eszközeikkel segítik a program bonyolultságának kezelését is. A különböző nyelvek lehetőségeikkel különböző programozási szemléleteket sugallnak, ezért is nagyon veszélyes az a sok helyen ma is elterjedt gyakorlat, amikor a programozás módszertanának oktatása az egyes programozási nyelveken keresztül, és nem azoktól függetlenül történik – ekkor ui. csak egy leszűkített kép alakulhat ki a programírás lehetőségeiről.

A programozás célja a *jó minőségű programtermék* előállítása, ezért a programozás oktatásának a feladat, és a feladatot megoldó program minél általánosabb megfogalmazásából kell kiindulnia [7], s e mellett, erre építve, kell megismertetnie a programozókat a különböző, konkrét nyelvi eszközökkel, amelyek támogatják elgondolásai megvalósításában. Ugyanakkor azonban, ahogy helytelen a módszertant egy-egy konkrét nyelven keresztül tanítani, hasonlóan zsákutcába vezet, ha azt mondjuk, hogy a használt programozási nyelv nem is fontos a módszertan szempontjából. Ez olyan – ahogy B. Meyer írja [17]–, mint „a madár szárnyak nélkül". A gondolat nem szeparálható el a kifejezés lehetőségeitől. Nem véletlen, hogy a programozásban egy nyelv sem lett egyeduralkodóvá, ahogy az sem, hogy egyre újabb és újabb programozási nyelveket terveznek, amelyek eszközeikkel egyre jobban támogatják a különböző módszertani elgondolások, követelmények gyakorlatba való átültetését.

A programozási nyelvek tervezőinek három problémával kell megküzdeniük [10]:

- a nyelvnek egyszerre kell absztrahálnia a számítógépből és az alkalmazásokból,
- jó jelölésrendszert kell biztosítania az algoritmus leírásához,
- a nyelv eszközként kell szolgáljon a program bonyolultságának a kezeléséhez.

Mielőtt magával az Ada programozási nyelvvel foglalkoznánk, nézzük meg, kik és mire használják a programozási nyelveket [13].

Amatőr és professzionális programozás

A mai időkre olyan mértékben megnőtt a programot író emberek száma, annyira kiszélesedett a programok és számítógépek felhasználási területe, hogy a programozás jól kivehetően két részre vált.

Az egyik esetben a programok rendszerint kicsik, általában olyan egyének írják, akiknek más a foglalkozásuk, pl. könyvelő, orvos, vegyészmérnök. Ők a számítógépet céljaik elérésében eszközként használják, s magával a géppel szemben sokszor nem is támasztanak nagy igényeket (esetleg egy kisebb személyi számítógép is megfelel). Ezek a programok gyakran rövid életűek, elképzelhető, hogy csak egyszer használják azokat.

Kevés karbantartásra van szükség, így nem is kell alapos dokumentációt hagyni róluk. A programok készítéséhez nagyon egyszerű szoftver eszközöket használnak, hiszen a kis méretek miatt a tesztelés triviális, és mivel a programot egy személy készíti, csak kevés adminisztrációra van szükség. A rövidéletűség miatt nem fontos, hogy a használt nyelv elterjedt vagy hosszú éveken keresztül stabil, szabványosított legyen. Az ezen a területen használt programozási nyelvek nagyon gyors fejlődésen mentek (mennek) át, és a szabványosítás a produktivitás ellen hatna. Ezeknek a nyelveknek fontos tulajdonságai a könnyű programírás lehetősége és a felhasználóval szembeni barátságosság. Az interaktív használat elengedhetetlen. Ilyen nyelvek pl. a BASIC, a FORTH, vagy a Pascal. Ezt *amatőr programozás*nak nevezhetjük.

A *professzionális programozás* ettől sok ponton eltér. Bár a feladatok igen különbözőek lehetnek, pl. adatfeldolgozási, rendszerprogramozási, tudományos számítási, folyamatirányítási feladatok, a programok készítésének természete itt minden feladat esetében ugyanolyan. Ezek a programok (vagy programrendszerek) rendszerint igen nagyok, és programozói csoportok készítik azokat. A csoportok tagjai foglalkozásuk szerint programozók, programtervezők. A számítógép hardverével szemben gyakran támasztanak a kapacitás, a megbízhatóság vagy egyéb minőségi jellemző tekintetében nagy igényeket. A programokat megfelelően dokumentálni kell, és a program élete során (ami több évtized is lehet) karbantartásra van szükség (pl. a program környezetének megváltozása esetén). Ezeket a programokat gyakran több – esetleg különböző típusú – gépen is használják. A professzionális programozást támogató nyelv minden változtatása csak erős ellenőrzés mellett vihető végbe. Egy ilyen nyelv fontos jellemzője a modularitás, a jól olvasható programszöveg. Ilyen létező nyelvek pl. az Ada, a C, a C++, a Java, az Eiffel, vagy a COBOL, a PL/I, a Modula-2 stb.[1]

[1] Az amatőr és professzionális elnevezés talán minőségi megkülönböztetést is sugall, de ilyenre nem gondoltunk.

A szoftver minőség szempontjai

A szoftver – termék, s mint minden termékre, erre is megfogalmazhatunk – ahogy megfogalmaztak sokan (például [17], [10], [15]) – különböző minőségi jellemzőket, követelményeket. A programozási módszertan egyik legfőbb célja, hogy elvi eszközt adjon a kezünkbe a jó minőségű programtermék létrehozásához. A programozási nyelv tervezését, illetve a már meglévő nyelvek értékelését feltétlenül befolyásolják módszertani megfontolásaink.

A következőkben először Bertrand Meyer munkája [17] alapján áttekintjük a „jó" szoftver jellemzőit. Ezután foglalkozunk csak a módszert támogató nyelvi eszközökkel – az Ada nyelven keresztül.

A szoftver minőségét számos tényező befolyásolja. Ezek egy részét – ilyenek pl. a megbízhatóság, a gyorsaság, vagy a könnyű használhatóság – alapvetően a program felhasználója érzékeli. Más jellemzők – például, hogy egyes részeit milyen könnyű egy más, de hasonló feladat megoldásához újra felhasználni – a programfejlesztőket érintik.

Helyesség

A programtermék *helyesség*e azt fejezi ki, hogy a program pontosan megoldja a feladatot, megfelel a kívánt specifikációnak.

Ez az első és legfontosabb kritérium, hiszen ha egy program nem úgy működik, ahogy kellene, akkor az egyéb szempontok nyilván nem sokat számítanak.

Ami itt alapvető: a pontos és minél teljesebb specifikáció.

Megbízhatóság

Egy programot *megbízható*nak nevezünk, ha helyes, és abnormális – a specifikációban nem leírt – helyzetekben sem történik katasztrófa, hanem valamilyen „ésszerű" működést produkál.

A fenti meghatározás jól tükrözi, hogy a megbízhatóság messze nem olyan precíz fogalom, mint a helyesség. Mondhatnánk ugyan, hogy ha precízebb, teljesebb specifikációt írtunk volna, akkor ez pontosan a helyességet jelentené, de a gyakorlat azt mutatja, hogy mindig vannak olyan esetek, amelyeket egy specifikáció nem ír le explicit módon. Ezért kiemelt jelentőségű a programtermék minősége szempontjából a megbízhatóság.

Karbantarthatóság

A *karbantarthatóság* annak mérőszáma, hogy milyen könnyű a programterméket a specifikáció változtatásához adaptálni.

Gyakran kéri a felhasználó a programtermék továbbfejlesztését, módosítását, új szabályozókhoz igazítását. Egyes felmérések szerint a szoftver költségek 70 %-át szoftver karbantartásra fordítják, így érthető, hogy ez a kritérium a program minőségét komolyan befolyásolja. (Ennek a követelménynek nagy programok, programrendszerek írásánál van jelentősége, hiszen kis programoknál általában semmilyen változtatás nem túl bonyolult.)

A karbantarthatóság növelése szempontjából a két legfontosabb alapelvnek a tervezési egyszerűséget és a decentralizációt (minél önállóbb modulok létrehozását) tekinthetjük.

Újrafelhasználhatóság

Az *újrafelhasználhatóság* a szoftver termékek azon képessége, hogy egészben vagy részben újra felhasználhatóak új alkalmazásokban.

Ez más, mint a karbantarthatóság, hiszen ott ugyanazon specifikáció módosult, míg most azt a tapasztalatot szeretnénk hasznosítani, hogy a szoftver rendszerek sok eleme közös mintákat követ, jó lenne elkerülni, hogy a már megoldott problémákat újra kidolgozzuk.

Ez a kérdés különösen fontos, nem is elsősorban egy egyedi programtermék előállításánál, hanem ha a szoftverkészítés globális optimalizációját tekintjük, hiszen minél inkább van lehetőségünk arra, hogy újrafelhasználható komponensek segítségével oldjuk meg feladatainkat, annál több energiánk marad a többi minőségi jellemző javítására is (azonos költség mellett).

Kompatibilitás

A *kompatibilitás* azt mutatja meg, hogy milyen könnyű a szoftver termékeket egymással kombinálni.

A programokat általában nem légüres térben fejlesztjük, a hatékonyságot nagyságrendekkel növelheti, ha a programok egymással összekapcsolhatóak. (Például a programok közti kommunikáció valamilyen standard konvenció alapján történik, mint a Unix-ban.)

Egyéb jellemzők

A programtermék minőségi jellemzői közül fontos odafigyelnünk a program hordozhatóságára, hatékonyságára, emberközeliségére, tesztelhetőségére, áttekinthetőségére stb. is.

A program *hordozhatóság*a annak mérőszáma, mennyire könnyű a programot más géphez, konfigurációhoz vagy operációs rendszerhez – általában más fizikai környezethez – átalakítani.

A program *hatékonyság*a a futási idővel és a felhasznált memória méretével arányos – minél gyorsabb, illetve minél kevesebb memóriát használ, annál hatékonyabb. (Ezek a követelmények sokszor egymás ellen hatnak, a gyorsabb futást sokszor nagyobb memóriaigény ellensúlyozza, és fordítva.)

A program *emberközelisége*, barátságossága a felhasználó számára rendkívül fontos: ez megköveteli, hogy az input logikus és egyszerű, az eredmények formája áttekinthető legyen.

A *tesztelhetőség*, *áttekinthetőség* a program karbantartói, fejlesztői számára fontos szempontok, ezek nélkül a program megbízhatósága sem garantálható.

Programok tervezési szempontjai

Az előző követelmények egy része – a helyesség és a megbízhatóság javítása – elsősorban a specifikációs eszközök fejlesztését igényli. Minél könnyebb belátni, hogy a programnyelvi leírás tényleg a specifikációnak megfelelő implementáció, annál könnyebben írhatunk helyes, illetve megbízható programokat. Itt a legnagyobb szerepe a specifikáció (típusinvariáns, elő-, és utófeltételek) leírását lehetővé tevő programnyelvi eszközöknek van – ezt támogatja például az Eiffel [17].

A követelmények másik csoportjának – gondoljunk itt elsősorban az újrafelhasználhatóság, karbantarthatóság, kompatibilitás szempontjaira – megvalósítását legjobban azzal támogathatjuk, ha programjainkat egymástól minél inkább független, jól definiált kapcsolatban álló programegységekként tervezzük. Ez az ún. *moduláris tervezés* alapja. (A modul itt nem programnyelvi fogalom, hanem a tervezés egysége.)

A modularitás kritériumai

Vegyük sorra [17] alapján, miket tekintünk a modularitás legfőbb szempontjainak!

Moduláris dekompozíció

A *moduláris dekompozíció* a feladat több egyszerűbb részfeladatra bontását jelenti, amely részfeladatok megoldása már egymástól függetlenül elvégezhető. Ennek segítségével csökkentjük a feladat bonyolultságát.

Általában a módszert ismételten alkalmazzuk, azaz az alrendszereket magukat is dekomponáljuk. Ezzel lehetővé tesszük azt is, hogy a feladat megoldásán egyszerre több ember is dolgozzon. A módszer egy fával ábrázolható, ahol a fa csomópontjai az egyes dekompozíciós lépéseknek felelnek meg. Pl. a felülről lefelé történő programtervezés alapvetően erre épül. Itt a tervezők a probléma absztrakt leírásából kiindulva a dekompozíció ismételt

alkalmazásával jutnak el a leírás egy olyan szintjére, amely már a konkrét programnyelven implementálható.

Moduláris kompozíció

Egy programtervezési módszer kielégíti ezt a kritériumot, ha olyan szoftver elemek létrehozását támogatja, amelyek szabadon kombinálhatók egymással.

Ez más szavakkal azt jelenti, hogy programjainkat minél inkább már meglévő programegységekből, mint építőkövekből szeretnénk felépíteni.

Míg a moduláris dekompozíció a program karbantarthatóságát támogatja, a *moduláris kompozíció* az újrafelhasználhatóságot.

Célunk, hogy a programkészítés minél nagyobb része már létező standard elemek kombinációja legyen. Már évtizedek óta fejlesztenek pl. a numerikus algoritmusok témájában alprogram-könyvtárakat.[2]

Moduláris érthetőség

Ez a kritérium azt jelenti, hogy a modulok önállóan is egy-egy értelmes egységet alkossanak, megértésükhöz minél kevesebb „szomszédos" modulra legyen szükség.

A programok karbantartása során lesz ez a szempont jelentős.

Moduláris folytonosság

Az elnevezést a folytonos függvények területe ihlette, azt jelenti, hogy a specifikáció „kis" változtatása esetén a programban is csak „kis" változtatásra legyen szükség. Ez azt is jelenti, hogy a „kis" változások egyes modulokat érintsenek csupán, s ne az egész program szerkezetét.

Sajnos az, hogy mit nevezünk „kis" változtatásnak a specifikációban, illetve a programban, nincs precízen definiálva, mégis, a fogalom intuitíven érthető.

Egy egyszerű példa: programjainkban numerikus, illetve szöveg konstansok helyett használjunk szimbolikus konstansokat – hiszen ekkor változás esetén csak egy helyen, a definícióban kell javítani.

Moduláris védelem

Célunk kell legyen a program egészének védelme az abnormális helyzetek hatásaitól. Egy hiba hatása egy – esetleg néhány – modulra korlátozódjon!

[2]az előző két kritérium független egymástól, sőt „egymás ellen" is dolgozhatnak, az ismételt dekompozíciós lépések eredményeként kapott modulok ui. igen sok esetben nagyon is speciális részproblémákat oldanak meg, s ezáltal nem igazán újrafelhasználhatóak.

Egy olyan módszer például, ami megköveteli, hogy minden modul, amelyik beolvas adatokat, egyúttal a felelős is az ellenőrzésükért, jó a moduláris védelem szempontjából.

Az előző kritériumhalmaz teljesítéséhez bizonyos elveket feltétlenül szem előtt kell tartanunk.

A modularitás alapelvei

A modulokat nyelvi egységek támogassák

A moduláris tervezés csak akkor lehet hatékony, ha a modulok illeszkednek a használt programozási (programtervezési, vagy specifikációs) nyelv szintaktikai egységeihez.

Programozási nyelv esetén ez az elv azt is jelenti, hogy a nyelvi modulok külön fordíthatóak kell legyenek. Bár ez teljesen technikai jellegű elvárás, de ha nem teljesül, komoly gondjaink lehetnek az előzőekben felsorolt kritériumok megvalósításával: például, ha egy rendszert jól elhatárolható részfeladatokra akarunk bontani (dekompozíció), akkor az a célszerű, ha ezek egyúttal különálló szintaktikai egységeket is képeznek. Másrészt, ha egy modul nem egy nyelvi egység, hanem több, vagy esetleg több egység bizonyos részeiből állítható össze, akkor a moduláris kompozíció válik lehetetlenné.

Amit a bevezetőben említettünk, itt is nyilvánvalóan megmutatkozik: megfelelő programnyelvi eszközök támogatása nélkül a legkiválóbb módszertani elvek is olyanok, mint „a madár szárnyak nélkül". A programnyelvi eszközöket csak részben lehet helyettesíteni a programozási stílussal és gondolkodásmóddal.

Kevés kapcsolat legyen

Minden modul minél kevesebb másik modullal kommunikáljon!

A modulok között sokféle kommunikáció lehetséges: a modulok hívhatják a másik modul által nyújtott szolgáltatásokat, megoszthatják adatszerkezeteiket stb. Ez az elv ezen kapcsolatok számát akarja korlátozni. Célul kell kitűzni, hogy egy `n` modulból álló rendszerben a modulkapcsolatok száma minél közelebb legyen az `n-1` minimális értékhez, és minél távolabb az `n*(n-1)/2` maximális értéktől.

Ez az elv voltaképpen a folytonosság és a védelem kritériumaiból ered: ha túl sok a kapcsolat a modulok között, akkor egy változtatás hatása vagy egy hiba terjedése is túl sok modult érint.

Gyenge legyen a kapcsolat

Ha két modul egyáltalán kommunikál egymással, akkor olyan kevés információt cseréljenek, amennyire csak lehetséges.

Ez az elv is a folytonosság és a védelem szempontjait veszi elsődlegesen figyelembe, hiszen minél kisebb a kapcsolat a modulok között, annál kevésbé hatnak ki az egyes modulokban bekövetkező változások a többi modulra, hasonlóképpen a hibák is annál kevésbé terjednek tovább.

Explicit interfész kell

Ez az alapelv azt mondja ki, hogy ha két modul kommunikál egymással, akkor annak ki kell derülnie legalább az egyikük szövegéből.

Ez az elv több kritériumot is érint: elsősorban a dekomponálhatóságét és a komponálhatóságét, hiszen mindkettő érdekében világosan jelezni kell minden külső kapcsolatot; a folytonosság kritériumának teljesíthetőségéhez is fontos, hogy az interfészből derüljön ki egyértelműen, hogy egy változtatás mely programegységekre lesz hatással; de talán a legjobban a moduláris érthetőség függ tőle – a modulok közti kapcsolatok nem láthatóak át világosan ennek teljesítése nélkül.

Információ elrejtése

Minden információ egy modulról rejtett (privát) kell legyen, kivéve, amit explicit módon nyilvánosnak deklaráltunk.

Ennek az elvnek az alkalmazása feltételezi, hogy minden modult az összes többi modul tervezője csak a hivatalos interfészen keresztül ismer. Ez azt jelenti, hogy minden modul alapvetően két részből áll: egy interfészből és egy privát, más modulokból semmilyen eszközzel nem elérhető részből. Ezt a lehetőséget a programnyelv kell biztosítsa. Természetesen elképzelhető, hogy az egész modul az interfészbe kerül, de az elv azt mondja ki, hogy ez legyen a „kivétel".

A folytonosság teljesíthetőségének alapvető feltétele ez. Ha ui. egy modulban csak a rejtett, privát részt változtatjuk, és az interfész változatlan marad, akkor a modullal kommunikáló más modulokat ez a változás egyáltalán nem érinti.

Nyitott és zárt modulok

Az eddigi szabályok egy részét a „hagyományos" programnyelvek is támogatják. Jó modulszerkezetet, információelrejtést megvalósíthatunk pl. a Modula-2-ben, vagy az Ada83-ban is. Az újrafelhasználhatóság követelménye azonban további támogatást is igényelne – ennek megfogalmazásához először két definíciót vezetünk be.

- Egy modult *zárt*nak nevezünk, ha más modulok számára egy jól definiált interfészen keresztül elérhető, a többi modul változatlan formában felhasználhatja.

Ez programnyelvi szempontból azt jelenti, hogy a zárt modul egy önállóan lefordított programegység, amelyet egy programkönyvtárban tárolhatunk és szükség szerint hozzászerkeszthetünk a programunkhoz.

- Egy modult *nyitott*nak nevezünk, ha még kiterjeszthető, ha az általa nyújtott szolgáltatások bővíthetőek vagy pl. hozzávehetünk további mezőket a benne levő adatszerkezetekhez, s ennek megfelelően módosíthatjuk a modul eddigi szolgáltatásait.

Jól látható, hogy a „zártság" és a „nyitottság" a hagyományos szemléletben egymásnak ellentmondó követelmények. A „hagyományos" programnyelvek általában lehetőséget adnak a zárt modulok készítésére, a felhasználó által tervezett absztrakt adattípus megvalósítására (modul, csomag stb.), de nem támogatják a nyitott modulokat. Az ún. objektum-orientált megközelítés egy új lehetőséget kínál majd: itt az alap, a kiindulás az osztály – ami az absztrakt adattípus megfelelője –, s egy új eszközt, az öröklődést adja arra, hogy már meglévő, lezárt osztályokból kiindulva – azokat nyitottként kezelve – hozzunk új osztályokat létre. (Erről lesz még szó majd a továbbiakban.)

Célunk, hogy bemutassuk, az Ada programozási nyelv tervezői milyen eszközöket választottak arra, hogy a professzionális programozókat támogassák a jó minőségű, megbízható szoftver készítésében.

1. Fejezet

Az Ada nyelv áttekintése

1.1. Az Ada kialakulásának történeti háttere

Az Ada programozási nyelv hatalmas nemzetközi erőfeszítés eredménye.

Az 1970-es évek elején az US DoD (United States Department of Defense) úgy döntött, hogy tennie kell a szoftver-költségek rohamos növekedése ellen. Már 1973-ban például a DoD szoftver-költsége 3 milliárd dollár körül volt, aminek 56 %-át az alap-szoftver jellegű rendszerek fejlesztése emésztette fel. Összehasonlításul: adatfeldolgozásra 19 %-ot, a tudományos programokra 5 %-ot költöttek.

A legnagyobb költség-megtakarítás nyilvánvalóan úgy érhető el, ha ezek közül az alap-szoftver rendszerek költségeit csökkentik. A rendszerprogramok készítéséhez nem alakult még ki igazán jól használható nyelv, úgy gondolták tehát, hogy jótékonyan hatna a költségekre ennek a területnek az egységesítése, egy új, nagy megbízhatóságú, alap-szoftver készítésére is alkalmas nyelvnek a kijelölése, illetve definiálása. Ennek érdekében szervezte meg az US DoD a HOL (High Order Language) elnevezésű munkát.

A hosszú távú cél elérésének első lépése a követelmények rögzítése és a létező nyelvek értékelése volt. A FORTRAN, COBOL, PL/I, HAL/S, TACPOL, CMS-2, CS-4, SPL/1, J3B, Algol 60, Algol 68, CORAL 66, Pascal, SIMULA 67, LIS, LTR, RTL/2, EUCLID, PDL2, PEARL, MORAL és EL-1 nyelvek értékelését publikálták. Az eredményt négy pontban foglalták össze:

1. Nincs olyan létező nyelv, ami egymagában megfelelne a célnak.
2. Egyetlen nyelvet kell kialakítani.
3. A követelmények definiálásának meg kell maradnia a „state of art” szintjén.
4. Az új nyelv fejlesztését a meglevő értékek felhasználásával kell végezni.

Az értékelt nyelveket három osztályba sorolták:

- *nem használható fel:* Ide tartoznak az elavult, vagy elhibázott nyelvek. Ezekről nem adtak részletesebb elemzést. Ebbe a kategóriába került például a FORTRAN és a CORAL 66.
- *megfelelő:* Ide kerültek azok a nyelvek, amelyek ugyan közvetlenül nem alkalmasak, de néhány elemük felhasználható. Ebbe a csoportba került pl. az RTL/2 és a LIS.
- *alapnyelvként használható:* Három ilyen nyelvet találtak, a Pascalt, a PL/I-et és az ALGOL 68-at, amiket az új nyelv tervezésének kiindulópontjául ajánlottak.

Széleskörű nemzetközi verseny után a CII Honeywell Bull franciaországi laboratóriumára bízták az új nyelv végleges kialakítását. Alapnyelvként a Pascalt választották.

Ebben az időben határozta el a DoD, hogy az új nyelvet Adának fogják hívni. Ada Augusta Byron – Lord Byron lánya, és Babbage asszisztense volt. ő volt a világon az első programozó.

Az Ada tervezésének [12] szempontjait a következőkben foglalhatjuk össze:

- **Biztonságra törekvés:** Legyenek a programok megbízhatóak, és legyen könnyű a karbantartásuk. (Inkább könnyen olvasható, mint könnyen írható legyen a program!) Ezért fontosnak tartották, hogy a nyelv:
 - erősen típusos legyen,
 - adjon lehetőséget az adatabsztrakció kezelésére,
 - külön fordítható programegységekből álljon,
 - legyenek eszközei a kivételes és hibás helyzetek kezelésére.
- Figyeltek az emberi programozóra is: a nyelv ne legyen túl bonyolult, s legyen a rendszere következetes – legyen összhangban azzal, amit a felhasználó normálisan „elvár”.
- Hatékony legyen.

Így született meg a nyelv 1983-as szabványa, az ún. Ada83. A nyelv alapja a Pascal, de néhány vonást átvett a következő nyelvekből is: Euclid, Lis, Mesa, Modula, Sue, Algol 68, Simula 67, Alphard, CLU.

Az azóta eltelt időben a következő főbb igények merültek fel:

- az interfészek problémája: – a programokat nem volt mindig könnyű más nyelven írt rendszerekhez illeszteni; s egyre inkább szükségesnek tűnt az eljárás paraméterek használata is (pl. a grafikus felhasználói felületeknél);
- a kiterjeszthetőség és újrafelhasználhatóság támogatása: – az Ada83 is ad részleges támogatást ezekre a csomag (`package`) és a sablon (`generic`) fogalmain keresztül, de más nyelvek tapasztalatai alapján úgy tűnik, hogy az objektum-orientált programozásban megmutatták az előnyeit annak, hogy egy programot úgy lehet kiterjeszteni, hogy nem kell módosítani a meglévő, bizonyítottan helyes (de legalábbis már többször letesztelt) komponenseket;
- szükséges egy rugalmasabb, hierarchikus könyvtár szerkezet;
- a párhuzamos programozást támogató eszközök fejlesztése.

A tervezők úgy érzik, hogy az Ada95, amelyet a fenti igények figyelembevételével az Ada83-ból kifejlesztettek, egy megbízható objektum orientált nyelv lett.

1.2. Az Ada rövid jellemzése

Egy Ada program egy vagy több programegységből áll, amelyek többnyire külön fordíthatóak.

Az Ada programegységei:

az alprogramok – végrehajtható algoritmusokat definiálnak, az alábbi két fajtájuk van:

- az eljárás – ez egy tevékenység-sorozat leírása, lehetnek paraméterei
- a függvény – egy érték kiszámítása és visszaadása

a csomagok – logikailag kapcsolatban álló entitások (alprogramok, típusok, konstansok és változók) gyűjteményeit definiálják,

a taszkok – párhuzamosan végrehajtható számításokat definiálnak.

a védett egységek – feladatuk a taszkok között megosztott adatok védett használatának koordinálása, erre definiálnak műveleteket, az egyszerű kölcsönös kizárás automatikusan definiált segítségükkel, de más védelem is definiálható. Egy védett belépési pont egy logikai kifejezés segítségével definiál egy belépési sorompót, ami igaz kell legyen, mielőtt a belépési ponthoz tartozó törzset végrehajtanánk.

a sablon (`generic`) egységek – típussal és alprogrammal is paraméterezhető, makró-szerű csomagot vagy alprogramot jelentenek.

A programegységekből (kivéve a taszkokat és a védett egységeket) kialakíthatóak önállóan fordítható egységek. Ezek között a főprogram egy olyan eljárás lehet, amelynek nincs paramétere.

Minden programegység alapértelmezés szerint két részből áll:

a specifikáció tartalmazza azt az információt, ami más egységek felé látható kell legyen

a törzs tartalmazza az implementációs részleteket – ez rejtett más programegységek felé.

A specifikációs rész és a törzs szétválasztása, valamint külön fordíthatósága lehetővé teszi nagymértékben független programkomponensek tervezését, írását és tesztelését.

A nyelv ad eszközöket a saját könyvtárak szervezésére. Minden könyvtár hierarchikusan van felépítve, ez lehetővé teszi, hogy individuális komponensekre dekomponáljuk a rendszert. Egy önállóan fordított program egység meg kell nevezze a szükséges könyvtári komponenseket.

A programegység törzse két részből áll:

deklarációs részből, ami definiálja a programegységben használt konstansokat, változókat, hibaeseményeket (*exception*) és programegységeket,

utasítássorozatból, ami definiálja a programegység végrehajtásának hatását.

A törzs végrehajtása ennek megfelelően két lépésben történik: az első lépés a deklaráció kiértékelése, a második az utasítássorozat végrehajtása. Megengedett a blokkszerkezet.

A megbízhatóságra törekvés egy fontos eszköze, hogy a végrehajtás során az ún. kivételes, vagy hibás helyzetekben a programozó dönthessen az eseményt követő tevékenységekről. A *kivétel*nek (`exception`) nevezett szituációk egy része előre definiált – így például `Constraint_Error` kivétel lép fel, ha valamilyen megszorítást megsértünk a programban (pl. indexhatárt túllépünk) –, de a programozó sajátmaga is definiálhat és kiválthat kivételeket. A kivételes esemény lekezelésére az egyes blokkok végén helyezhetünk el megfelelő utasításokat, s mód van arra is, hogy itt a jelenséget esetleg korrigáljuk, vagy legalább „mentsük, ami menthető" elven, pl. a használt állományokat lezárjuk, s ezután jelezzük a hibát a magasabb szintekre.

Az Ada *erősen típusos* nyelv. Minden objektum rendelkezik egy, a fordítás során egyértelműen megállapítható típussal, ami egy típusértékhalmazt és egy alkalmazható művelethalmazt jelent. A típusok megfelelő használata ellenőrzött.

Vannak a nyelvnek elemi típusai, mint a felsorolási típus, a különböző numerikus és mutató(`access`) típusok, valamint összetett típusai – ide tartoznak a rekord és a tömb típusok is.

A numerikus számításoknál a nyelv megkülönbözteti a pontos és a közelítő számításokat, az előbbihez egész típusok használatát ajánlja, az utóbbihoz vagy a fixpontos típusokat (abszolút hibahatárral), vagy a lebegőpontos típusokat (relatív hibahatárral).

Az összetett típusok strukturált objektumok definícióit teszik lehetővé. A tömbök – azonos típusú elemekből állnak, a komponensekre indexükkel hivatkozhatunk. A rekordok különböző típusú elemekből állhatnak, a komponenseket nevük azonosítja. A taszk és a védett (protected) típusok is az összetett típusok közé tartoznak.

A rekord, taszk és a védett típusoknak lehetnek diszkriminánsnak nevezett speciális komponensei – ezekkel paraméterezhetjük ezeket a típusokat. Például a variáns rekord szerkezet is így definiálható.

A mutató (`access`) típusok a láncolt adatszerkezetek létrehozását támogatják, egy mutató típus egy értéke egy referenciát reprezentál egy deklarált, vagy allokátorral létrehozott objektumra. Alprogramokra mutató típusokat is definiálhatunk.

Az adatabsztrakciót a felhasználó által definiált típusok ún. átlátszatlan (`private`) nézete támogatja. A felhasználó a csomag (`package`) programegység specifikációs részében megadhatja az új típus értékhalmazát és a műveletek specifikációját – azaz az összes, a típus használói számára fontos információt, s elrejtheti a típus reprezentációját a csomag specifikációs részének a végén, s a műveletek implementációját a csomag törzsében, s ezek így csak a csomag és a gyerekcsomagok számára lesznek elérhetőek.

Minden típusból származtatással új típus hozható létre. Egy típus a leszármazottaival együtt egy származtatási osztályt alkot. Osztályszintű műveletek definiálhatóak, amelyek a származtatási osztály bármely típusából elfogadnak paramétert. A rekord és az átlátszatlan típusokra a leszármazottak a szülő típus kiterjesztései is lehetnek. Az Ada95 a *jelölt* (`tagged`) típusok lehetőségét adja, hogy a leszármazási osztályon belül az operandus dinamikus típusa futási időben azonosítható legyen. Ha egy jelölt típus egy műveletét olyan operandusra alkalmazzuk, amelynek a dinamikus típusa csak futási időben ismert, akkor futásidejű összekapcsolás történik.

Az altípusképzéssel a típusértékhalmaz szűkíthető. Altípusokat használhatunk skaláris típusok részintervallumaira, tömböknél a lehetséges indexértékekre vonatkozó megszorításra, rekordoknál a diszkriminánsok speciális értékeire.

Reprezentációs záradékokat használhatunk a típusok és a gép közötti leképezés leírására, például a felhasználó specifikálhatja, hogy egy adott típus objektumait hány biten tárolja.

Az Ada nyelv tartalmaz egy szabványos könyvtárat, az input-output, a szövegkezelés, a véletlenszám generátor stb. tartozik bele, ahogy majd látni fogjuk.

Általában lefordított Ada egységek egy speciális könyvtárban, egy adatbázisban helyezkednek el, mely kezdetben csak előredefiniált csomagokat tartalmaz, s ezekhez illesztheti hozzá a felhasználó a maga könyvtári egységeit. Egy Ada program kifejlesztése a könyvtári egységek adatbázisának felépítését jelenti.

Kiemelkedő jelentőségű az Ada sablon (`generic`) konstrukciója: A makró fogalom általánosításaként ezek a programegységek eszközt adnak arra, hogy egy csomagot vagy alprogramot mintaként, típussal, alprogamokkal, objektumokkal, sőt csomagokkal is paraméterezhetően írjunk le. Így például nem szükséges külön-külön megvalósítani a `Verem` típust egészekre, valósakra, karakterekre stb., hanem leírhatjuk a tárolt elem típusától függetlenül, s az egyes konkrét esetekben létrehozhatjuk belőle a megfelelő új adattípust.

Ezt az általánosítási lehetőséget jól támogatja az Ada egy érdekessége: Nem csak típusokra, hanem típusosztályokra is definiál műveleteket – például tetszőleges skaláris (felsorolási, egész vagy valós) `T` típusnak lekérdezhetjük az első értékét a `T'First` attribútummal, vagy bármely `V` objektum memóriabeli címét megtudhatjuk a `V'Address` attribútum segítségével.

Az Ada a nyelv különböző variánsainak kialakulása ellen is védekezik. A nyelv szabványa tartalmazza azt a kitételt, hogy *csak az a fordítóprogram használhatja az „Ada fordítóprogram" megnevezést, ami a nyelv minden szabályát pontosan alkalmazza, s ami nem vezet be sem módosításokat, sem kiterjesztéseket.* Azt, hogy egy fordítóprogram viselheti-e az „Ada fordítóprogram" címkét, egy New Yorkban felállított intézetben döntik el. Az ide elküldött fordítóprogram validálása úgy történik, hogy lefordítanak és lefuttatnak egy több ezer programból álló tesztsorozatot.

2. Fejezet

Lexikális elemek

Az Ada program egy vagy több fordítási egységből áll. A fordítási egységek lexikális elemek sorozatai. Lexikális elemek megadott szabályoknak megfelelő karaktersorozatok, amelyeket egymástól elhatárolójelek választanak el. Az Ada95 program forrásszövegében általában az ISO-10646 szabványban felsorolt 65536 karakter bármelyikét használhatjuk, bár pl. azonosítókban csak az ISO-8859 szabvány által is definiált 193 grafikus karakter fordulhat elő[1].

A nyelv megkülönböztet *grafikus*[2], *formázó*[3] és egyéb vezérlőkaraktereket. Grafikus karakterek a számjegyek, a szóköz, az ún. azonosítóbetűk és a speciális karakterek. Az *azonosítóbetű*k halmaza viszonylag szűk, a Latin-1 ábécé kis- és nagybetűiből áll, beleértve a nyugat-európai nyelvekben előforduló ékezetes betűket is. A *speciális karakter*ek közé tartozik az összes többi grafikus karakter, amely az ISO-10646 szabványban szerepel, pl.:
`" # & ' ( ) * + , - . / : ; < = > _ [ ] { } ő ű |` stb.[4]

Az Ada az azonosítókban és az alapszavakban nem tesz különbséget a kis- és nagybetűk között. Az azonosító minden karaktere szignifikáns. A nyelvben – a modern gyakorlatnak megfelelően – az alapszavakat külön jelölni nem kell, de ezek kötött szavak, azonosítóként nem használhatóak. Az Adában 69 alapszó van:

`abort`	`case`	`exit`	`not`	`range`	`terminate`
`abs`	`constant`	`for`	`null`	`record`	`then`

[1]A nyelv definíciója ezen szabványoktól való eltéréseket is megenged, ha az így kapott programszöveg információveszteség nélkül a szabványos karakterkészlet segítségével is felírható és viszont. Az Ada95 programok forrásszövegének belső ábrázolása és grafikus megjelenítése egyaránt implementációfüggő.

[2]A szabványos karakterkészlet minden karaktere grafikus a 0-1F,7F-9F,FFFE-FFFF kódú karakterek kivételével.

[3]A formázó karakterek a következők: vízszintes és függőleges tabulátor, kocsivissza, soremelés, lapdobás karakter.

[4]Ahol az Ada95 megengedi, ott az `ő`,`ű` betűket használjuk. A könyvben szereplő példaprogramokban ahol a Latin-2 karaktereket az Ada nem támogatja, ott helyettük `ô`-t, ill. `û` -t írunk.

```
abstract  declare    function  of         rem       type
accept    delay      generic   or         renames   until
access    delta      goto      others     requeue   use
aliased   digits     if        out        return    when
all       do         in        package    reverse   while
and       else       is        pragma     select    with
array     elsif      limited   private    separate  xor
at        end        loop      procedure  subtype
begin     entry      mod       protected  tagged
body      exception  new       raise      task
```

További megszorítás, hogy az azonosítókban csak aláhúzásjel és azonosítóbetűk használhatóak és nem kezdődhetnek aláhúzásjellel. Vannak a nyelvben előre definiált jelentésű szavak is (pl. a típusok felépítésében használt `Integer`, `True`, `String` stb.), ezek újradefiniálhatóak. Az Adában az a szokás, hogy az alapszavakat kisbetűkkel írják, az azonosítókat pedig nagybetűkkel kezdik.

A számoknak két alapformája van: a decimális és az alapszámmal felírt forma. Mindkét esetben megadható a szám exponenciális alakban is, ebben az esetben a kitevőt egy `E` betű vezeti be. A számrendszer alapszáma megelőzi a szám számjegyeit, amelyek sorozatát egy-egy `#` jel vezet be és zár le. A számrendszer alapszáma kettőtől tizenhatig bármelyik szám lehet, de maga az alapszám és az exponens mindig decimális alakban adott. A számok tagolására használhatjuk az aláhúzásjelet, amelynek a szám értéke szempontjából nincs jelentősége.

```
123_456_789 -- decimális egész szám
1E6         -- decimális egész exponenciális alakban
2#1011#     -- bináris egész szám
3.14159     -- decimális valós
16#F.FF#E2  -- hexadecimális valós exponenciális alak
2#1.1111_1111_111#E11 -- a kitevőhöz tartozó alap a
                      -- számrendszer alapszáma, tehát a
                      -- két utóbbi érték egyaránt 4095.0
```

A valós számokat az különbözteti meg az egész számoktól, hogy tizedespontot tartalmaznak. Valósak exponenciális alakjában negatív kitevő is szerepelhet.

Az Ada karakter- és sztringliterálokban grafikus karaktereket használhatunk. A karakterliterálokat úgy írjuk, hogy a karaktert aposztrófok közé zárjuk. A sztringliterálokat idézőjelek közé tesszük, és nem nyúlhatnak túl a sor végén. A nem grafikus karaktereket karakteres konstans formájában a predefinit `Standard` csomag definiálja.

```
'A' 'a' '''         -- ez egy nagy és kis A betű és egy aposztróf
""                  -- üres sztring
""""                -- egy idézőjelet tartalmazó sztring
"abc" & 'd' & "ef"  -- ez a konkatenáció fordítási időben
                    -- értékelődik ki
ASCII.ESC & "[9B"
```

Az elemi szintaktikai egységek (alapszavak, azonosítók, literálok stb.) közé tetszőlegesen sok szóköz elhelyezhető, de ezek belsejében nem lehet szóköz. (Nem lehet tehát szóköz a több karakterrel kifejezett szimbólumok, pl. `:=`, `/=`, `=>` stb. belsejében sem.) Lexikális elemek között általában *elhatároló jel* a szóköz, a formázó karakterek bármelyike, a sorvége, továbbá egyes speciális karakterek és karakterkombinációk:

`& ' ( ) * + , - . / : ; < = > => .. ** := /= >= <= << >> <>`

Ezek akkor határolnak lexikális elemeket, ha nem megjegyzés vagy sztringliterál belsejében fordulnak elő. A vízszintes tabulátor csak akkor nem elhatároló jel, ha megjegyzésben fordul elő. Egyetlen azonosító, illetve egy sor megengedett legnagyobb hossza az Ada95 implementációjától függ, de legalább 200 karakter.

A megjegyzést két mínuszjel vezeti be, és a sor végéig tart.

`-- ez egy megjegyzés.`

Megjegyzés a program bármely sorában elhelyezhető. A megjegyzésnek nincs hatása sem a program szintaktikus helyességére, sem a jelentésére.

2.1. Feladatok

2.1. Melyek a helyes azonosítók az alábbiak közül?
`Halmaz_Osztály, 1_Alma, Ős_Típus, %arány, Út, _Sebesség.`

2.2. Mi az értéke az alábbi literáloknak?
`16#E.AB#E1, 2#1001, 8#770E-1.`

3. Fejezet

Típusok és kifejezések

3.1. Típusok

Egy típust a lehetséges értékeivel és az ezekkel végezhető alapvető műveletekkel jellemezhetünk.

Például a beépített `Integer` típust az egész számok (az adott implementációtól függő rész-) halmaza:

```
..., -3, -2, -1, 0, 1, 2, 3, ...
```

és a szokásos műveletek:

```
+, -, *, stb.
```

jellemeznek.

A *deklaráció* olyan nyelvi szerkezet, amely egy entitás egy nézetéhez hozzárendel egy nevet. A deklaráció lehet explicit vagy implicit. Minden deklaráció tartalmazza az entitás nézetének definícióját. A *nézet* pedig tartalmazza az entitás megnevezését, valamint a hozzáférést meghatározó további jellemzőket (például egy objektum elérési módját vagy egy alprogram formális paramétereit). Gyakran a deklaráció tartalmazza az entitás definícióját is. A deklaráció kiértékelése futási időben történik.

A nyelv pontosan definiálja egy adott deklaráció hatáskörét: a programszöveg azon részét, amelyben a deklarált entitásra hivatkozhatunk. A deklaráció tehát létrehozza a nevet, a megfelelő nézetet és általában magát az entitást is. A deklarált entitásokhoz általában egy azonosítót rendelünk hozzá, de például a felsorolási típusoknál ez lehet egy karakterliterál, a függvények neve pedig lehet egy operátor szimbólum is.

```
type Mat_Char is ('+', '-', '*', '/');
function "+" (A, B: Mátrix) return Mátrix;
```

Ha a programban egy azonosítóra hivatkozunk, akkor annak a szövegben már deklaráltnak kell lennie.

A típusoknak —néhány speciális kivételtől eltekintve— saját nevük van, melyet egy típusdeklarációban vezetünk be. A beépített típusok —ilyen például az `Integer` — a `Standard` előre definiált csomagban vannak deklarálva. Minden típusdeklaráció egy, a meglévő típusoktól logikailag élesen elváló új típust vezet be. Egy típus csak önmagával ekvivalens.

Két különböző típus típusértékei is különbözőek, még ha az értékek lexikális formája meg is egyezik. Az, hogy ekkor egy érték éppen melyik típushoz tartozik, az a szövegösszefüggésből derül ki. Azt az elvet, hogy ugyanaz a lexikális alak több, különböző dolgot is jelenthet, *átlapolásnak* nevezzük.

Egy típus értékeit nem adhatjuk értékül más típusú változónak. Ez az erős típusosság alapszabálya. Minden objektumnak egyetlen típusa van. Az erős típusosság hatalmas segítség a programozónak helyes programok készítéséhez, mivel nagyon sok hibát már fordítás közben kiszűrhet.

A típus deklarációja a `type` alapszóval kezdődik, a típushoz rendelt név, majd az `is` kulcsszó következik, ezután a típus definíciója, végül a záró pontosvessző.

```
type Típus_név is ...;
```

Az `is` és a `;` közötti rész írja le, hogy milyen értékek tartoznak a típushoz. Egy konkrét példa a következő:

```
type Szín is (Piros, Zöld, Kék);
```

Ez egy felsorolási típus (lásd a 3.3.2. alfejezetet). A deklaráció létrehozza a `Szín` típust. Továbbá azt is leírja, hogy csak három érték van ebben a típusban, melyeket a `Piros`, `Zöld` és `Kék` azonosítókkal jelölünk.

Az objektumok deklarációja egy kissé eltér a típusok deklarációjától. Egy típushoz tartozó objektumokat a következőképpen hozhatunk létre:

```
Sz: Szín;
```

Ha egy típust absztraktnak deklarálunk az `abstact` kulcsszóval, akkor ennek a típusnak nem lehetnek saját objektumai. Az objektumoknak kezdeti értéket is adhatunk:

```
Sz: Szín := Piros;
```

vagy egy konstanst is deklarálhatunk:

```
Minta: constant Szín := Piros;
```

Az erős típusosság miatt az értékadás bal, illetve jobb oldalán csak ugyanolyan típusú változók szerepelhetnek. Így a következő értékadás helytelen:

```
I: Integer;
Sz: Szín;
...
I := Sz;
```

Az Ada elemi típusai a skalár típusok (diszkrét és valós) és a mutató típusok. A diszkrét típusok lehetnek felsorolási vagy egész típusok. A valós típusok lebegőpontosak vagy fixpontosak.

Az összetett típusok a tömbök, a rekordok, a jelölt rekordok, a taszk típusok, és a védett típusok.

A nyelv által definiált típusosztályok hierarchikus struktúráját a következőképpen ábrázolhatjuk:

```
Elemi típus
 - skalár
       - diszkrét
              - felsorolási
                   - karakter
                   - logikai
                   - egyéb felsorolási
              - egész
                   - elöjeles egész
                   - moduló típusok
       - valós
              - lebegöpontos
              - fixpontos
                   - tetszöleges fixpontos
                   - decimális fixpontos
 - mutató (dinamikus)
       - objektumra mutató
       - alprogramra mutató
Összetett típusok
 - tömb
       - string
       - más tömbök
 - jelöletlen rekord
 - jelölt rekord
 - taszk
 - védett
```

Egy T típus művelete minden olyan alprogram, amely T típusú értéket ad eredményül, vagy T típusú operandusa van, vagy pedig a T típust jelölő mutató paramétere van. Az előre definiált operátorok, vagy a nyelv által definiált egyéb műveletek, mint például a :=, vagy az elem-e vizsgálat a típus előre definiált műveletei.

A típus primitív műveletei a típus előre definiált operátorai, valamint a felhasználó által definiált primitív alprogramok. Specifikus típusok primitív műveleteit a következőképpen definiáljuk: származtatott típus esetén az öröklött, felhasználó által definiált alprogramok; felsorolási típus esetén a felsorolási literálok (amelyek paraméter nélküli függvények); egy csomag specifikációjában deklarált típus esetén, bármely olyan alprogram, mely ugyanebben a csomagban van deklarálva és a típuson hajt végre műveletet; valamint bármely alprogram, amely ugyanabban a deklarációs részben van deklarálva, mint a típus és felüldefiniálja a típus más implicite deklarált primitív alprogramját.

3.1.1. Altípusok

Ebben a részben bevezetjük az altípusok és a korlátozások fogalmát. Egy típus *altípusa* a típus értékhalmazának részhalmazát jelöli. A részhalmazt korlátozásokkal adjuk meg. Hasonlóan más részhalmazokhoz, az altípus is lehet az egész halmaz. Nincs mód viszont a típus műveleteinek korlátozására. Az altípusra alkalmazható a típus összes művelete, a korlátozás csak az értékhalmazra vonatkozik. [1]

Példaként tegyük fel, hogy dátumokkal akarunk foglalkozni és tudjuk, hogy a hónap napjai az `1 .. 31` intervallumba esnek. Ehhez deklarálhatunk egy altípust:

```
subtype Napok is Integer range 1 .. 31;
```

Itt a `range` fenntartott szót követő `1 .. 31` egy intervallum korlátozás. Ezután az altípus azonosítójával deklarálhatunk változókat és konstansokat pontosan úgy, mint egy típus azonosítójával:

```
N: Napok;
```

Biztosak lehetünk benne, hogy az `N` változó csak egész értékeket kaphat és csak az `1 .. 31` intervallumból. A fordítóprogram garantálja ennek teljesülését. Ha a korlátozás nem teljesül, akkor `Constraint_Error` hibát fog kiváltani.

[1] Ada83-ban az altípus típusát alaptípusnak nevezték. Mi is használjuk időnként ezt a megfogalmazást a magyarázatokban. Ada95-ben a típus nevét úgy tekintjük, mint a típus egy altípusának a neve. Ezt az altípust nevezzük első altípusnak. Az Ada Referencia Kézikönyvben (ARM) az altípus (subtype) elnevezést találjuk minden alkalommal, ahol típusról van szó.

Fontos megjegyezni, hogy egy altípus deklarációja nem vezet be új típust. Az előző példában szereplő `N` típusa `Integer`, így a következő szintaktikusan helyes lesz:

```
N: Napok;
I: Integer;
...
N := I;
```

Végrehajtás közben, ha az `I` értéke benne van az `1 .. 31` intervallumban, akkor az értékadás helyes, ha nincs benne, akkor `Constraint_Error` kivétel fog fellépni. A másik irányú értékadás:

```
I := N;
```

természetesen mindig működni fog.

Nem minden esetben szükséges explicit altípust bevezetni egy korlát megadásához. Például írhattuk volna azt is, hogy

```
N: Integer range 1 .. 31;
```

Az is lehetséges, hogy az altípus nem vezet be korlátozást:

```
subtype Elemek is Integer;
```

Ebben az esetben a `Napok` értékhalmaza megegyezik az `Integer` típus értékhalmazával, az ilyen deklarációkkal a kód olvashatóságát növeljük.

Altípust létrehozhatunk már létező altípus segítségével is:

```
subtype Február_Napok is Napok range 1 .. 29;
```

A további korlátozások ki kell elégítsék a meglévő korlátozásokat, pl.:

```
M: Napok range 0 .. 10;
```

helytelen lenne és `Constraint_Error` hibához vezetne, mivel a `Napok` értékhalmaza eredetileg `1`-től kezdődik és nem `0`-tól.

Egy másik példa Adában a `Natural` altípus:

```
subtype Natural is Integer range 0 .. Integer'Last;
```

Ezután a `Positive` típust kétféleképpen is deklarálhatjuk:

```
subtype Positive is Integer range 1 .. Integer'Last;
   -- vagy
subtype Positive is Natural range 1 .. Integer'Last;
   -- vagy 1 .. Natural'Last
```

ahol `Integer'Last` az egész típus legnagyobb értéke, lásd az 3.3.1. (attribútumok) alfejezetben.

Az eddigi példákban olyan korlátozásokat használtunk, amelyeknek határai statikusak voltak. Ez nem mindig van így, általában a korlátozásokat tetszőleges kifejezéssel adhatjuk meg, és így egy altípus értékeinek halmaza nem feltétlenül statikus (vagyis fordítás közben ismert).

Összegzésképpen tehát megállapíthatjuk, hogy egy altípus nem vezet be új típust, hanem egy meglévő típusnak vagy altípusnak adhatunk új nevet opcionális korlátozással.

Az altípusok képzése azért fontos, mert az Ada nyelvben számos olyan hely van, ahol nem írhatunk explicit korlátozásokat és ilyen esetekben egy altípust kell létrehozni, használni.

Az altípusok helyes használata két előnnyel jár: egyrészt biztosítja, hogy a programban sok hibát korábban észreveszünk, mivel kizárja, hogy egy változónak nem megfelelő értéket adjunk. Másrészt növeli a program végrehajtásának a hatékonyságát (például tömbök indexei esetén).

3.1.2. Származtatott típusok

Az Ada nemcsak különböző típusok összekeverése ellen nyújt védelmet, hanem a szintaxis szintjén megkülönbözteti az ugyanabból a típusból létrehozott új típusok objektumait is. Ezt az eszközt, amivel új típust hozunk létre már létező típusból, *származtatásnak* nevezzük. Egy típus teljesen definiált kell legyen, mielőtt bármit is származtatnánk belőle. Például:

```
type Hosszúság is new Float;
```

A származtatott típus átveszi az eredeti típus struktúráját, értékhalmazát és műveleteit, átveszi a kezdeti értékeit is. Fontos azonban, hogy a két típus nem ekvivalens. Ha például a következőképpen határozzuk meg a `Terület` típust:

```
type Terület is new Float;
```

és deklaráljuk a

```
H: Hosszúság;
T: Terület;
```

változókat, akkor a `H := T;` utasítás hibás, mert a baloldal és a jobboldal típusa nem egyezik meg.

A származtatás az altípusképzéssel is kombinálható:

```
type Napok is new Integer range 1 .. 31;
```

mely ekvivalens a következővel:

```
type Hónap_Nap is new Integer;
subtype Napok is Hónap_Nap range 1 .. 31;
```

Az eredeti típus értékeit átkonvertálhatjuk a származtatott típusba és viszont, konverziós függvényként a típus nevét kell használni.

```
type Szín is (Piros, Sárga, Zöld);
type Lámpa is new Szín;
L: Lámpa;
S: Szín;
```

A fenti deklarálások után a `L := Lámpa(S)` és a `S := Szín(L)` konverziók helyesek.

A származtatott típus jellemzői: bármely típusosztály, amely tartalmazza a szülő típust, tartalmazza a származtatott típust is.

Ha a szülő típus egy elemi típus vagy egy tömb típus, akkor a származtatott típus lehetséges típusértékei a szülő típus lehetséges típusértékeinek másolatai. Skalár típus esetén a származtatott típus bázis altípusa megegyezik a szülő típus bázis altípusával. A tömbtől eltérő összetett típusok esetén lásd a 12. fejezetet.

Ha a szülő típus korlátozott (`limited`), akkor a származtatott típus is az. A szülő típus bármely előre definiált műveletének van egy megfelelője a származtatott típus számára.

A szülő típusnak a felhasználó által definiált bármely primitív alprogramjára, amely létezett a származtatott típus definiálásakor, létezik egy neki megfelelő öröklött primitív alprogramja a származtatott típusnak és ez ugyanazzal a névvel rendelkezik. A felhasználó által definiált „=" operátor is öröklődik, kivéve, ha a származtatott típus egy nem korlátozott rekord kiterjesztés, ekkor ugyanis ezt be kell építeni.

Ha a szülő típus logikai típus, akkor a származtatott típus előre definiált relációs operátorai logikai típusú eredményt adnak vissza.

Ha az `abstract` szót használjuk egy típus származtatásánál, akkor a típusnak nem lehet saját objektuma.

A típusokat *származtatási osztályokba* csoportosíthatjuk. Egy származtatott típus *közvetlen leszármazottja* a szülő típusnak és *közvetett leszármazottja* bármely olyan típusnak, amelytől a szülő osztályt származtattuk.

Egy `T` típusra *a típusok származtatási osztálya* az a halmaz, mely tartalmazza a `T`-t (az osztály *gyökértípusát*), minden olyan típust, amelyet a `T`-ből származtattunk, valamint a `T`-hez kapcsolódó univerzális vagy osztályszintű típust.

A típusokat a következőképpen csoportosíthatjuk: *specifikus*, *osztályszintű* vagy *univerzális típus*.

A *specifikus típust* vagy a típusdeklarációban definiáljuk, vagy egy formális típusdeklarációban, vagy pedig egy objektumhoz rendelt teljes típusdeklarációban.

Az *osztályszintű típus* minden olyan származtatási osztályra létrejön, amelynek gyökértípusa jelölt típusú. Ha adott a `T` jelölt típus és `S` az altípusa, akkor `S'Class` a `T'Class` jelölt osztályszintű típus megfelelő altípusát jelképezi. Az ilyen típusokat osztályszintűnek nevezzük, mert ha egy formális paramétert `T'Class` típusúnak definiálunk, akkor az aktuális paraméter a `T`-ben gyökerező származtatási osztály bármely típusához tartozhat. A `T'Class` osztályszintű típus típusértékhalmaza a `T`-ben gyökerező származtatási fa összes specifikus típusa típusértékhalmazának az uniója. Az osztályszintű típusoknak nincs saját primitív műveletük.

Az *univerzális típusokat* az egész, a valós és a fixpontos osztályokhoz csatolták, *universal_integer*, *universal_real*, valamint *universal_fixed* elnevezésekkel.

Ezek megfelelnek a fent említett numerikus típusok osztályszintű típusainak. Ha egy formális paraméter univerzális típusú, akkor az aktuális paraméter az adott osztály bármelyik típusához tartozhat.

Az egész osztálynak és a valós osztálynak —az univerzális típus mellett— van egy-egy speciális gyökér típusa: a *root_integer*, valamint a *root_real*.

Minden numerikus típus származtatott típus, implicit módon a neki megfelelő gyökértípusból származik.

Az osztályszintű típus, illetve az univerzális típus lefedi az osztályához tartozó összes típust. Egy specifikus típus csak önmagát fedi le.

Egy `T2` típust a `T1` leszármazottjának nevezünk, ha a `T2` megegyezik a `T1` típussal vagy ha `T2` egy közvetlen vagy közvetett származtatással keletkezett a `T1`-ből. A `T2'Class` a `T1` leszármazottja, ha `T2` leszármazottja a `T1`-nek. Ha `T2` a `T1` leszármazottja, akkor `T1` a `T2` őse. Hasonlóan, egy univerzális típus a saját osztályának megfelelő gyökértípus leszármazottja.

3.2. Objektum deklarációk és értékadások

Egy típus értékeit az adott típusú objektumokban tárolhatjuk. Objektum a változó, melynek értéke megváltozhat a program futása közben, a konstans, amely teljes élettartamára megtartja a felvett kezdeti értékét, a formális paraméter, a ciklusváltozó, a tömbelem, a résztömb, a taszk és a védett objektum.

Egy változót a deklarációjával vezethetünk be. A deklaráció a változó nevéből (azonosítójából) és típusának nevéből áll. A két nevet kettőspont választja el. Ezt követheti opcionálisan egy `:=` jel és egy kezdőérték. A deklarációt pontosvesszővel zárjuk le. Például:

```
I: Integer;
P: Integer := 38;
```

Az első deklaráció bevezeti az `I` `Integer` típusú változót, de nem ad neki kezdeti értéket. A második a `P` változót vezeti be, kezdőértéke `38` lesz.

Egy deklarációban több változót is bevezethetünk, vesszővel elválasztva:

```
I, J, K: Integer;
P, Q, R: Integer := 38;
```

Itt a második sorban `P`, `Q` és `R` kezdeti értéke egyaránt `38` lesz.

Ha egy változónak nem adunk kezdőértéket, vigyázzunk arra, hogy ne hivatkozzunk rá, amíg valamilyen módon értéket nem kap. Ha mégis felhasználjuk az ilyen változót, akkor a program viselkedése nem definiált. Kaphatunk például `Constraint_Error` kivételt, de az is lehet, hogy a program tovább fut, a változóban „véletlenszerű" érték (társzemét) lesz.

Egy változónak gyakran értékadó utasítással adunk értéket. Itt a változó nevét := követi, majd egy kifejezés, ami megadja az új értéket. Az utasítást pontosvessző zárja le. Például

```
I := 38;
-- és
P := Q + R;
```

egyaránt helyes értékadó utasítások, új értéket helyeznek `I`-be, illetve `P`-be, felülírva az ott talált értéket.

Az értékadás := jelét bármilyen kifejezés követheti, feltéve, hogy az eredmény típusa megegyezik a változó típusával. A kifejezések írásának pontos szabályait később tárgyaljuk, de annyit itt megjegyezhetünk, hogy a kifejezések —hasonlóan más nyelvekhez— konstansokból, változókból, műveletekből és függvényekből állhatnak.

Érdemes összehasonlítani a kezdőértékadást is tartalmazó deklarációt és az értékadó utasítást. Mindkét esetben a := jelet használjuk, és mindkét helyen tetszőlegesen bonyolult kifejezést írhatunk a jobb oldalra. Egy fontos különbség, hogy deklarálhatunk egyszerre több változót is, ezekhez adhatunk kezdőértéket ugyanazzal a kifejezéssel, nem adhatjuk viszont ugyanazt az értéket egyetlen értékadó utasítással több változónak.

Fontos megjegyezni, hogy a többszörös deklaráció (akár kezdőértékadással is), csupán csak egy rövidítés. Vagyis

```
A, B: Integer := F;
```

valójában azt jelenti, hogy

```
A: Integer := F;
B: Integer := F;
```

Tehát `F` kétszer értékelődik ki az egy sorba írt esetben is. Ha `F` például egy `Integer` értéket visszaadó paraméter nélküli függvény, amely minden hívásakor egy véletlenszámot ad vissza, akkor `A` és `B` különböző értékeket kaphat. Leggyakrabban azonban konstans kifejezéseket használunk többszörös deklarációk kezdőértékadásánál.

A konstansokat a változókhoz hasonlóan deklaráljuk, a `constant` szó hozzáadásával. Ugyancsak konstans: az `in` típusú paraméter, a diszkrimináns és a ciklusváltozó.

```
Pi: constant Float := 3.1415;
```

Szám típusoknál (és csak itt) elhagyhatjuk a típus nevét a konstans deklarációjából.

```
Pi: constant := 3.1415;
```

Ilyen deklarációval *univerzális számkonstansokat* hozhatunk létre, azaz nevet adunk egy számnak. A név ugyanúgy használható, mint maga a szám, amit jelöl, például különböző típusú kifejezésekben. Az egész és valós típusok közötti megkülönböztetést a kezdeti érték alapján tesszük (van-e benne tizedespont). Soha nem hagyhatjuk el viszont a típus megjelölését változók deklarációjából, még akkor sem, ha adunk neki kezdőértéket is.

Fontos, hogy különbség van a típussal rendelkező konstansok és a típustalan univerzális számkonstansok kezdőértékadó kifejezése között: előbbieknél bármilyen kifejezést használhatunk, a kifejezés futási időben fog kiértékelődni. Az univerzális számkonstansoknál viszont a kezdőértékadó kifejezésnek konstansnak kell lennie és ezt a fordító már fordítás közben kiértékeli.

Az Ada élesen elválasztja a deklarációkat, melyek új azonosítókat vezetnek be és az utasításokat. Egy azonosító deklarációjának meg kell előznie az azonosítóval műveleteket végző utasításokat. Így a deklarációk és az utasítások a programszövegben is más-más helyen fordulhatnak elő. A legegyszerűbb szerkezet, amelyben deklarációk és utasítások is előfordulnak a blokk utasítás. Egy blokk utasítás a `declare` kulcsszóból, deklarációkból, a `begin` kulcsszóból, utasításokból, az `end` kulcsszóból és a lezáró pontosvesszőből áll:

```
declare
   I: Integer := 0; -- deklarációk
   J: Integer;
begin
   J := I+1;        -- utasítások
   I := J-1;
end;
```

A blokk utasítás maga is utasítás, így a belsejében szereplő utasítások egyike lehet egy újabb blokk utasítás. A blokkok tetszőleges mélységben egymásba ágyazhatók.

Mivel a blokk is egy utasítás, ugyanúgy végrehajtható, mint bármely más utasítás. Ez úgy történik, hogy a deklarációk (a `declare` és a `begin` kulcsszavak közötti rész) kiértékelődnek egymás után, majd a törzs utasításai (a `begin` és az `end` között) végrehajtódnak sorrendben egymás után. Figyeljük meg az elnevezésbeli megkülönböztetést: a deklarációk kiértékelődnek, míg az utasítások végrehajtódnak. Egy deklaráció kiértékelése létrehozza a deklarált objektumot, kiértékeli a kezdőértéket megadó kifejezést, majd hozzárendeli a kezdőértéket az objektumhoz.

A blokk végén a blokkban deklarált összes objektum megszűnik.

A fenti példa létrehozza `I`-t és `J`-t, megváltoztatja az értéküket, majd véget ér anélkül, hogy bármire is felhasználtuk volna `I` és `J` értékét.

A kezdeti értékadáshoz használt objektumoknak a felhasználás pillanatában már létezniük kell. Deklarálhatjuk ugyanabban a deklarációs részben, de az objektum deklarációjának meg kell előznie az objektumra hivatkozást:

```
declare
   I: Integer := 0;
   K: Integer := I;
begin
...
end;
```

A deklarációk kiértékelésének sorrendje lényeges, a leírt sorrendben történik. [2]

Meg kell különböztetnünk a hatáskör és a láthatóság fogalmát. A hatáskör a programszövegnek az a része, ahol az entitásnak valamilyen hatása van. Egy blokk utasítás esetén egy objektum hatásköre az objektum deklarációjának kezdetétől a blokk végéig tart. A láthatóság viszont azt jelenti, hogy a programszöveg egy adott pontján hivatkozhatunk rá egy adott névvel.

3.3. Skalár típusok

Az Ada skalár típusai a *diszkrét* (ezen belül a *felsorolási* és *egész* típusok) és a *valós* típusok.

A skalár típusok értékei rendezettek, értékeire alkalmazhatjuk a relációs operátorokat: `<`, `<=`, `>`, `>=`, `in`, `not in`. Az `=` és a `/=` reláció minden nem

[2] Ha nagyon sok csomagot használunk a programban és a csomagok használják egymás deklarációit, akkor előfordulhat, hogy egy olyan csomagban szereplő objektumra hivatkozunk, amelynek deklarációs része még nem értékelődött ki. Ilyenkor használhatjuk az `Elaborate` pragmát a csomagok kiértékelési sorrendjének megadására.

korlátozott típusú objektumra alkalmazható. Figyeljük meg, hogy a „nem egyenlő" operátort `/=` formában írjuk. Egy skalár típus részhalmazát az *intervallum* (`range`) segítségével kaphatjuk meg. Az intervallumoknak van alsó és felső határa és a következő a szintaxisa:

```
range Alsó_határ .. Felsô_határ
```

Segítségével a következőképpen képezhetünk részhalmazokat:

```
subtype Számjegyek is Integer range 0 .. 9;
```

ami tulajdonképpen egy altípust jelöl. Üres intervallumot kapunk akkor, ha az intervallum felső határa kisebb, mint az alsó határa. Az alsó, illetve a felső határ értékeit megadhatjuk kifejezések segítségével is, azzal a feltétellel, hogy a kifejezés típusa megegyezzen az intervallum elemeinek típusával. Egy intervallum benne van egy másik intervallumban, ha az intervallum minden egyes értéke eleme a másik intervallumnak. Egy diszkrét típusú elem esetén, a pozíciószám mutatja, hogy az elem hányadik a típusértékek között. Az első elem pozíciószáma 0.

3.3.1. Attribútumok

Az egyes típusosztályok rendelkeznek ún. *attribútumokkal*, *típusműveletekkel*, amelyeket az osztály minden típusára használhatunk. Minden `S` skalár altípusra léteznek a következő attribútumok:

`S'First` - az `S` értékhalmazának alsó korlátját jelöli.

`S'Last` - az `S` értékhalmazának felső korlátját jelöli.

`S'Range` - ekvivalens az `S'First .. S'Last` által meghatározott intervallummal.

`S'Base` - az `S` megszorítás nélküli altípusát jelöli, melyet az `S` *bázistípusának* nevezünk.

`S'Min` - a következő függvényt jelöli:

```
function S'Min(Bal, Jobb: S'Base) return S'Base
```

A függvény a paraméterként kapott két érték közül a kisebbet adja vissza.

`S'Max` - ugyanaz mint az `S'Min` csak a két érték közül a nagyobbat adja vissza.

`S'Succ` - a következő függvény:

```
function S'Succ(X: S'Base) return S'Base
```

A függvény felsorolási típus esetén az `X` érték pozíciószámánál eggyel nagyobb pozíciószámú `S'Base`-beli értéket adja vissza. Ha nincs ilyen szám, akkor `Constraint_Error` hibát vált ki. Ha az argumentum egész szám, akkor az argumentumot növeli 1-gyel; ha fixpontos szám, akkor a small értéket adja hozzá; ha lebegőpontos szám, akkor azt a gépi számot adja vissza, amely közvetlenül az argumentum után következik.

`S'Pred` - a fenti függvényhez hasonló, felsorolási típus esetén az `X` érték pozíciószámánál eggyel kisebb pozíciószámú értéket adja vissza. Analóg módon, ha az argumentum egész szám, akkor 1-gyel kisebb értéket ad vissza; ha fixpontos szám, akkor kivonja a small értéket; ha lebegőpontos szám akkor azt a gépi számot adja vissza, ami közvetlenül az argumentum előtt található.

`S'Wide_Image` - a következő konverziós függvényt jelöli:

```
function S'Wide_Image(X: S'Base) return Wide_String
```

A függvény az `X` argumentumot látható karakterláncként adja vissza. Az egész érték képe a megfelelő decimális literál (aláhúzás, vezető nullák, záró szóközök nélkül), melyet a szóköz vagy a '-' jel előz meg. A felsorolási típus képe vagy a megfelelő azonosító nagybetűs alakban vagy a megfelelő karakterliterál. Nem grafikus karakterekre nagybetűs formában kiírja a nyelv vagy az implementáció által definiált nevét (pl: NUL). Lebegőpontos értékre azt a decimális valós literált írja ki, amely legjobban megközelíti a számot(kerekítve). A kép tartalmazza a '-' vagy a szóköz karaktert, 1 számjegyet (ez nem 0, ha a szám különbözik nullától), a tizedespontot, `S'Digits`-1 számjegyet, az 'E' karaktert, majd a kitevő előjelét és a kitevő számjegyeit (ha a `S'Signed_Zeros True`, akkor a 0 előjele '-'. Fixpontos értékre a kiírás tartalmazza az előjelet vagy szóközt, az egészrész számjegyeit, a tizedespontot és a tizedesrész számjegyeit (`S'Aft` számjegyet).

`S'Image` - ugyanaz mint az `S'Wide_Image` azzal a különbséggel, hogy `X` értékét sztringként adja vissza. Ha egy karakter nem grafikus, akkor a karakter megjelenítése implementációfüggő.

`S'Wide_Width` - az `S'Wide_Image` által visszaadott `Wide_String` maximális hosszúságát jelöli. Nullát ad vissza, ha az altípus üres típus (*universal_integer* típusú 0-t).

`S'Width` - az `S'Image` által visszaadott `String` maximális hosszúságát jelöli.

`S'Wide_Value` - a következő függvény:

```
function S'Wide_Value(X: Wide_String) return S'Base
```

Az `X` argumentumot átkonvertálja `S'Base` típusúra (nem veszi figyelembe a kezdeti szóközöket).

`S'Value` - ugyanaz, mint az `S'Wide_Value`, azonban az `X` paraméter `String` típusú.

3.3.2. A diszkrét típusok

Az Ada diszkrét típusai a felsorolási és az egész típusok. A diszkrét típusok a skalár típusok attribútumai mellett még két fontos attribútummal rendelkeznek.

`S'Pos` - a következő függvényt jelöli:

```
function S'Pos(X: S'Base) return Integer
```

Az `X` érték pozíciószámát adja vissza.

`S'Val` - a következő függvényt jelöli:

```
function S'Val(X: universal_integer) return S'Base
```

Ha paraméterként egy pozíciószámot írunk, akkor a típusértékek közül a neki megfelelő értéket adja vissza. Ha nem létezik ilyen érték, akkor `Constraint_Error` kivétel lép fel.

Érvényesek a következő egyenlőségek:

```
S'Pos(S'Succ(x)) = S'Pos(x) + 1
S'Pos(S'Pred(x)) = S'Pos(x) - 1
S'Succ(S'Pred(x)) = x
S'Val(S'Pos(x)) = x
S'Pos(S'Val(x)) = x
S'Value(S'Image(x)) = x
```

Ha egy típus első értékére alkalmazzuk a `Pred`, illetve a típus utolsó értékére alkalmazzuk a `Succ` műveletet, akkor `Constraint_Error` kivételt fogunk kapni.

Felsorolási típusok

A felsorolási típus definíciójában a típusértékeket azonosítóikkal soroljuk fel. A sorrend határozza meg a nagyságrendi viszonyaikat. Az azonosítók a programban literálként használhatóak.

A következőkben felsorolási típusok deklarációjára mutatunk néhány példát:

```
type Szín is (piros, zöld, kék);
type Napok is (Hétfô, Kedd, Szerda, Csütörtök,
   Péntek, Szombat, Vasárnap);
type Szóló is (Egyedül);
type Szigetlakó is (Robinson, Péntek);
```

Itt átlapolásra is láthatunk példát: a `Péntek` felsorolási literál lehet `Napok` vagy `Szigetlakó` típusú. (Emlékezzünk, hogy Adában a kis és nagybetűket azonosítókban nem különböztetjük meg.)

`Péntek` mindkét jelentése látható a programszövegben, a második deklaráció nem rejti el az első deklarációt. Még akkor sem, ha ugyanabban a deklarációs részben, vagy az egyiket egy beágyazott blokk deklarációs részében deklaráljuk. Általában a szövegösszefüggésből egyértelmű, hogy melyiket akarjuk használni. Ha mégsem, akkor minősíthetjük a literált úgy, hogy zárójelbe tesszük a nevét, elé a típusa nevét és egy aposztrófot írunk:

```
Nap'(Péntek)
Szigetlakó'(Péntek)
```

A `Péntek` azonosítót használhatjuk két különböző felsorolási típusban mint literál értéket, de nem használhatjuk egyszerre felsorolási típus literáljaként és változó azonosítójaként. A felsorolási literált akár még alprogram nevével is átlapolhatjuk.

Egy felsorolási típusban nincs felső határ az elemek számára, de legalább egy elemnek lennie kell, az üres felsorolási típus nem megengedett.

A felsorolási típusokra alkalmazhatunk korlátozásokat intervallumok segítségével:

```
range Alsó_határ_kifejezés .. Felsô_határ_kifejezés
```

Ez azt adja meg, hogy az értékek halmaza az alsó határral kezdődik és a felső határig tart. Például:

```
subtype Munkanap is Napok range Hétfô .. Péntek;
```

Ezután írhatjuk azt, hogy:

```
M: Munkanap;
```

vagy azt is, hogy

```
M: Napok range Hétfô .. Péntek;
```

Itt `M` értéke nem lehet `Szombat` vagy `Vasárnap`.

Ha az alsó határ nagyobb, mint a felső határ, akkor üres intervallumot kapunk:

```
subtype Színtelen is Szín range Zöld .. Piros;
```

Figyeljük meg, hogy a `Szóló` típusból nem tudunk üres intervallumot létrehozni, mivel összesen egyetlen érték van benne.
Az attribútumokat felsorolási típusokra a következőképpen használhatjuk:

```
Szín'First = Piros
Munkanap'Last = Péntek
Szín'Succ(Piros) = Zöld
Napok'Pred(Péntek) = Csütörtök
```

A zárójelben tetszőleges, adott típusú kifejezés lehet. Példák a `Pos` és a `Val` használatára:

```
Szín'Pos(Piros)= 0
Szín'Pos(Zöld) = 1
Szín'Pos(Kék) = 2
Szín'Val(0) = Piros
Napok'Val(6) = Vasárnap
```

Ha az intervallumon kívül eső pozíciószámot adunk meg, mint például:

```
Szóló'Val(1)
```

akkor `Constraint_Error` fog fellépni.

A felsorolási típusok esetén a `<`, `<=`, `>` és `>=` operátorok eredményét a deklaráció sorrendje definiálja. Ezért a `T'Pos(X) < T'Pos(Y)` reláció és a `X < Y` reláció ugyanazt az eredményt adja.

Példák relációs operátorokra:

```
Piros < Zöld           = True
Szerda >= Csütörtök    = False
```

A karakter típusok

Két, a `Standard` csomagban előre definiált karakter típus a `Character` és a `Wide_Character`. A `Character` 256 kódpozíciója megfelel az ISO-10646 Basic Multilingual Plane (BMP) Row 00 (Latin-1)-nak. Bármely grafikus karakternek megfelel egy karakterliterál. A nem grafikus karaktereknek a nyelv által meghatározott neve van. A `Wide_Character` a fent említett szabvány 65536 kódpozíciója. Az utolsó két pozíció kivételével bármely más értéknek megfelel egy grafikus karakter. Ezek az input-output műveleteknél használt típusok, ezekben találhatjuk meg a nyomtatható karaktereket. A karaktereket a program szövegében aposztrófok közé írjuk. Például:

```
type Magánhangzók is ('A', 'E', 'I', 'O', 'U');
```

A Boolean típus

A `Boolean` típus egy előre definiált felsorolási típus:

```
type Boolean is (False, True);
```

`Boolean` értékeket használunk logikai kifejezésekben, mint például az if utasítás feltételében. `Boolean` értékeket állítanak elő az =, /=, <, <=, > és >= operátorok. Például írhatunk ilyen kódrészletet, mint

```
if Ma = Vasárnap then
   Holnap := Hétfô;
else
   Holnap := Nap'Succ(Ma);
end if;
```

A `Boolean` típus esetén, a `Succ` attribútumot így használhatjuk:

```
Boolean'Succ(False) = True
```

Azt is írhatjuk, hogy:

```
subtype Mindig is Boolean range True .. True;
```

A `Boolean` típus operátorai:

`not` egyoperandusú operátor, `True`-ból `False`-t állít elő és fordítva.

`and` kétoperandusú operátor, az eredmény `True`, ha mindkét operandusa `True`, különben `False`.

`or` kétoperandusú operátor, az eredmény `True`, ha az egyik, vagy mindkét operandus `True`. `False`, ha mindkét operandus `False`.

`xor` szintén bináris operátor, az eredmény `True`, ha az egyik operátor `True` a másik `False`. Úgy is mondhatjuk, hogy akkor `True`, ha az operandusok különbözőek.

A legmagasabb precedenciája (lásd a 59. oldalon) a `not`-nak van (ugyanaz mint az `abs` és `**` operátoroknak). Az `and`, `or` és `xor`-nak pedig a legkisebb a precedenciájuk. Létezik még két úgynevezett *rövidzár operátor*: az `and then` és az `or else`.

Ugyanazt az értéket adják, mint az `and`, illetve az `or`, de ha az egyik operandus kiértékelése után már eldönthető az eredmény, akkor a másik operandust nem értékelik ki (míg az `and` és az `or` mindig kiértékeli mindkét operandust). A gyakorlatban majdnem mindig célszerűbb az `and then` és az `or else` használata.

Ezeknek a kétoperandusú operátoroknak megegyezik a precedenciája, és ez kisebb, mint bármelyik más operátoré. Például kisebb a precedenciájuk mint az `=`, `/=`, `<`, `<=`, `>`, `>=` relációs operátoroké. Ez Pascal-ban és más nyelvekben nem így van, tehát Adában nem szükséges zárójeleket használni olyan kifejezésekben, mint

```
P < Q and I = J
```

Habár a precedenciájuk megegyezik, nem keverhetjük egy kifejezésen belül az `and`, `or`, `xor` operátorokat zárójelek használata nélkül (ellentétben a `+` vagy `-` operátorokkal):

```
B and C or D      -- helytelen
I  +  J - K       -- helyes
```

Azt kell írnunk helyette, hogy

```
B and (C or D)    vagy   (B and C) or D
```

így kihangsúlyozva, hogy melyik jelentést akarjuk használni.

Más programozási nyelvekben az `and` és az `or` általában különböző precedenciájú, amit a programozó könnyen összekeverhet és pont a nem kívánt módon használhatja. Hogy ezt elkerüljük, ezért az Ada ezeknek azonos precedenciát adott és kétséges esetben zárójeleket kell használnunk.

Természetesen ugyanazt az operátort használhatjuk többször is egy kifejezésben:

```
B and C and D   -- megengedett
```

A kiértékelés – mint máshol is – balról jobbra történik.

Figyeljünk oda a `not` operátorra. A precedenciája magasabb, mint az `and` az `or` vagy az `xor` precedenciája, így

```
not A or B  jelentése (not A) or B  nem pedig  not (A or B)
```

A `Boolean` változókat és konstansokat a szokásos módon deklarálhatjuk és kezelhetjük:

```
Baj: Boolean;
Jelzô: Szín;
...
Baj := Jelzô = Piros;
```

A baj változó `True`, ha a `Jelzô = Piros`. Ezután írhatjuk például azt, hogy

```
if Baj then
   Állítsd_meg_a_vonatot;
end if;
```

Figyeljük meg, hogy nem szükséges ezt írnunk:

```
if Baj = True then ...
   else ...
```

Az egész számok típusai

Az egész számok típusait két csoportba oszthatjuk: előjeles és előjel nélküli (vagy moduló) típusokra.

Ezenkívül léteznek az alábbi, az implementációtól függő előre definiált egész típusok: `Short_Integer`, `Long_Integer`, `Short_Short_Integer`, illetve a `Long_Long_Integer`. Nem kötelező, hogy egy implementáció ezeket tartalmazza, de ha beépíti akkor előírás, hogy például az `Integer` típus tartalmazza a $-2**15+1$.. $+2**15-1$ intervallumot, vagy a `Long_Integer` a $-2**31+1$.. $+2**31-1$ intervallumot. Ugyanakkor az `Integer` típus nem lehet hosszabb, mint a `Long_Integer`.

Egy egész típus definíciója implicite az előre definiált névtelen *root_integer* típusból származik, mely a `System.Min_Int .. System.Max_Int` intervallum.

Ahogy azt már láttuk, az `Integer` típusra is bevezethetünk korlátozást a `range` segítségével.

```
P: Integer range 1 .. I+J;
```

A határértékek nem feltétlenül konstans értékek. Az intervallum lehet üres is, például a fenti példában, ha `I+J` értéke 0. Az üres intervallumok haszontalannak tűnhetnek, de gyakran szükség van rájuk például korlátozások megadásához; az üres intervallum elkerülése sokkal bonyolultabbá tenné a programot.

Egy egész típust deklarálhatunk a következőképpen is:

```
type Int is range 1 .. 20;
```

Az `Int` típus elemei a névtelen *universal_integer* egész típus elemei, `1..20` korlátozással. Az `Integer` típus legkisebb értéke az `Integer'First`, a legnagyobb értéke pedig `Integer'Last`.

Az `Integer'First` értéke implementációfüggő, de bizonyosan negatív.

Egy kettes komplemens számítógépen `Integer'Last-1` lesz, míg egyes komplemens gépen `-Integer'Last` lesz az értéke. Például egy minimális 16 bites kettes komplemens implementációban:

```
Integer'First = -32767
Integer'Last = 32767
```

Természetesen mindig jobb azt írni, hogy `Integer'Last`, mint a konkrét számértéket az adott rendszeren ahol dolgozunk — a program hordozhatósága érdekében.

Két hasznos altípus, amivel már találkoztunk:

```
subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

Ezeket is a `Standard` előre definiált csomagban deklarálja a nyelv.

A `First` és `Last` attribútumok altípusokra is használhatóak, így például:

```
Positive'First = 1
Natural'Last = Integer'Last
```

A moduló típusok előjel nélküli egészek ciklikus aritmetikával. Tegyük fel például, hogy 8 bites (byte) aritmetikát szeretnénk végezni:

```
type Elôjel_nélküli_byte is mod 256;
```

Az `Elôjel_nélküli_byte` típus lehetséges értékei 0 .. 255 -ig terjednek. A szokásos aritmetikai műveleteket használhatjuk, de valamennyi műveletet moduló 256 végezzük és így nem léphet fel túlcsordulás. A moduló típus modulusának statikusnak kell lennie. Nem kell, hogy a kettő hatványa legyen, bár ez gyakori. Hasznos lehet például nagy prímszámot modulusnak használni.

`S'Modulus` adja vissza egy moduló típus modulusát, mely *universal_integer* típusú.

Az egész típusokra egyaránt használhatjuk a modern programozási nyelvek műveleteit:

`+`, `-` ezek egy- vagy kétoperandusú operátorok. Az egyoperandusú + nem csinál semmit, az egyoperandusú - megváltoztatja az előjelet. Ha bináris operátor, akkor mindkét operandusnak egész típusúnak kell lennie, az eredmény is egész típusú lesz és a szokásos összeadás vagy kivonás történik.

`*` Szorzás, mindkét operandusnak egész típusúnak kell lennie, az eredmény szintén egész típusú lesz.

`/` Osztás, mindkét operandusnak egész típusúnak kell lennie, az eredmény szintén egész típusú lesz. Az egészek osztása 0 felé csonkítja az eredményt.

`rem` Maradékképzés, mindkét operandus egész típusú kell legyen, az eredmény is egész típusú. Az osztás maradékát adja vissza értékül, `A rem B` előjele megegyezik `A` előjelével és `A=(A/B) * B + (A rem B)`.

`mod` Moduló, mindét operandus egész típusú kell legyen, az eredmény is egész. Ez a matematikai moduló művelet, `A mod B` előjele megegyezik `B` előjelével és `A = B * N + (A mod B)`, ahol `N` egy egész szám.

`abs` Egyoperandusú operátor, az operandusa egy egész szám. Eredményként az abszolút értéket fogjuk kapni. Vagyis ha az operandus pozitív vagy 0 akkor az operandust, ha az operandus negatív, akkor a megfelelő pozitív értéket kapjuk vissza.

`**` Hatványozás, az első operandust felemeli a második operandus hatványra. Az első operandus egy egész típusú, a második pozitív egész vagy 0 kell legyen.

Az egész típusú változók összehasonlítására használhatjuk a relációs operátorokat: `=`, `/=`, `<`, `<=`, `>`, `>=`. Ezek `Boolean` típusú eredményt adnak vissza (`True` vagy `False`). A `rem` és `mod` különbségét a következő példák szemléltetik:

A	B	A/B	A rem B	A mod B
12	5	2	2	2
-12	5	-2	-2	3
12	-5	-2	2	-3
-12	-5	2	-2	-2

```
   13    5    2      3      3
  -13    5   -2     -3      2
   13   -5   -2      3     -2
  -13   -5    2     -3     -3
```

3.3.3. A valós típusok

A valós értékek hibájának pontosabb definiálásához az Ada új fogalmakat vezetett be. Minden valós típus esetén definiálja azoknak az értékeknek a halmazát, amelyet a nyelv minden implementációjának pontosan kell ábrázolnia. Ezeket az értékeket a típus *modellszámainak* nevezzük. A modellszámok csak a típustól függenek, az implementációtól nem. Ugyanakkor minden valós típushoz definiálja a *biztonságos számok* halmazát, amely a típusnak az adott implementációban pontosan ábrázolható értékeit jelenti. A biztonságos számok halmaza lehet bővebb a modellszámok halmazánál.

A típusértékek egy olyan intervallumát, amelynek mindkét végpontja modellszám, *modell intervallumnak* nevezzük. Minden valós számhoz hozzárendelhetünk egy modell intervallumot, azt a legkisebb modell intervallumot, ami a számot tartalmazza. A műveletek és a relációk a modell intervallumokon vannak definiálva. Egy művelet eredménye mindig az a legkisebb modell intervallum, ami tartalmazza azt a legkisebb és legnagyobb matematikailag pontos eredményt, amit az operandus intervallumok értékeire kaphatunk. Ha az eredményül kapott modell intervallum valamelyik határa abszolút értékben nagyobb mint a legnagyobb biztonságos szám, akkor `Constraint_Error` hibát vált ki.

Kétféle valós típust különböztetünk meg: a lebegőpontos és a fixpontos valós típust.

Lebegőpontos típusok

A lebegőpontos számok egy fix hosszúságú *mantisszából* és egy előjeles egész *exponensből* állnak. *Normalizált formának* hívjuk a valós számnak azt az alakját, amelyben a mantissza egész része nulla és az első tizedesjegy nullától különböző (kivéve a 0.0).

Adában az előre definiált lebegőpontos típus a `Float`, mely implementációfüggő. Ezért a programokban célszerű, hogy az ebből származtatott típusokat használjuk.

```
type Real is new Float;
```

A lebegőpontos valós típus pontosságát a mantissza ábrázolásához szükséges decimális jegyek számának megadásával definiáljuk. Jelöljük ezt D-vel. A modellszámok mantisszájának ábrázolásához B bitre van szükség, ahol: $B = D * log(10)/log(2) + 1$. A modellszámok kitevőjének értéktartománya a $-4 * B..4 * B$ intervallum.

```
type Real is new Float digits 8;
```

vagy röviden:

```
type Real is digits 7;
```

A valós típust is korlátozhatjuk, megszorítást adhatunk az értékhalmazára:

```
type Real is digits 8;
subtype Valószínûség is Real range 0.0 .. 1.0;
P,Q: Valószínûség;
```

Az egész típushoz hasonlóan létezhetnek különböző `Float` implementációk: `Short_Float`, `Long_Float` stb. A lebegőpontos típus attribútuma:

`S'Digits` - azt az értéket jelöli, amelyet a típus deklarálásakor adtunk meg a `digits` kulcsszó után.

Fixpontos típusok

A fixpontos számokat fix számú számjeggyel és egy képzeletbeli tizedesponttal ábrázoljuk. Ha a törtrész k számjegyből áll, akkor a $10 * *(-k)$ értéket a számhoz tartozó *deltának* nevezzük. Mivel a számok ábrázolása a kettes számrendszerben történik, a típus értékei nem az $x * 10 * *(-k)$ számok, hanem $x * 2 * *(-p)$ számok, ahol p az a legkisebb pozitív egész szám, amelyre fennáll a $2 * *(-p) <= 10 * *(-k)$ egyenlőtlenség. A $2 * *(-p)$ számot a típus *small értékének* nevezzük.

Példák:

```
type Valós is delta 2.0**(-7) range -1.0 .. 1.0;
type Fix is delta 0.01 range 0.0 .. 1.0;
```

A `Fix` típusú valós számok két tizedes számjegyet tartalmaznak. A 0.01 pontosság helyett a *small* értéke 1/128 lesz. Ezért, ha a 0.05 értéket a `Fix` típusba konvertáljuk, akkor a 0.046... értéket kapjuk. A hibák elkerülése miatt célszerű reprezentációs előírással definiálni a típushoz a small értéket (lásd a 17. fejezetet):

```
type Fix is delta 0.01 range 0.0 .. 1.0;
for Fix'Small use 1.0/2**(Word_Length - 1)
```

Egy tizedes fixpontos altípus létrehozását a `digits` típusú korlátozás segítségével tehetjük meg:

```
type Money is delta 0.01 digits 15;
subtype Salary is Money digits 10;
-- Money'Last=10.0**13 - 0.01, Salary'Last=10.0**8 - 0.01
```

A fixpontos típusok attribútumai:

`S'Small` - a típushoz tartozó *small* értéket jelöli.

`S'Delta` - a *delta* számot jelöli.

`S'Fore` - a tizedespont előtti karakterek minimális számát jelöli kiírás esetén. A reprezentáció nem tartalmaz kitevőt, de tartalmazhat '-' előjelet vagy szóközt.

`S'Aft` - a decimális számjegyek száma (az a legkisebb n, ahol n pozitív egész szám, amelyre igaz a $(10 * *n) *$ `S'Delta` $>= 1$ egyenlőtlenség.

`S'Digits` - a tizedes fixpont típus digits értékét jelöli, mely megfelel a reprezentáció decimális számjegyeinek számának.

`S'Scale` - azt az n számot jelöli, amely az `S'Delta` $= 10 * *(-n)$ egyenlőséget teljesíti.

`S'Round` - a paraméterként kapott értéket kerekíti.

A valós típusok műveletei:

`+`, `-` ezek egy- vagy kétoperandusú operátorok és az összeadást, illetve a kivonást végzik el.

$*$ Szorzás, mindkét operandusnak valós típusúnak kell lennie, az eredmény szintén valós típusú lesz.

`/` Osztás, mindkét operandusnak valós típusúnak kell lennie, az eredmény szintén ilyen típusú lesz.

$**$ A hatványozás esetén az első operandus valós típusú, míg a második egész típusú.

A relációs operátorokat a valós típusokra is használhatjuk.

Megjegyzések

Általában kevert módú aritmetika nem lehetséges, vagyis nem lehet például egy egész számot egy lebegőpontos számhoz hozzáadni, mivel a két operandusnak ugyanolyan típusúnak kell lennie. Átalakíthatunk viszont egy értéket `Integer` típusúból `Float` típusúvá, ha a kívánt típus vagy altípus neve után írjuk a konvertálandó kifejezést zárójelben. (Emlékezzünk rá, hogy Adában csak kerek zárójeleket használunk.) Például ha

```
I: Integer := 3;
F: Float := 5.6;
-- akkor nem írhatjuk azt, hogy
F := I+F;
-- hanem azt kell írnunk, hogy
F := Float(I) + F;
```

ami lebegőpontosan adja össze az értékeket és az eredmény 8.6 lesz, vagy

```
I + Integer(F)
```

ez egész összeadást használ és az eredmény 9 lesz.

A `Float`-ból `Integer` felé történő konverzió mindig kerekít (nem pedig levágja a tizedesrészt) így 1.4-ből 1 lesz és 1.6-ból 2 lesz. Két egész közötti fél mindig a 0-tól távolabbi egész számra kerekítődik, vagyis 1.5-ből 2 lesz, -1.5-ből -2.

Mint más nyelvekben, így az Adában is az operátoroknak különböző precedencia szintjük van, ami felülbírálható zárójelek használatával. Az azonos precedenciájú operátorokat balról jobbra alkalmazzuk. Egy zárójelben lévő részkifejezést nyilván előbb kiértékelünk, mint ahogy felhasználjuk az értékét. Az eddig megismert operátorok precedencia szintje növekvő sorrendben:

```
and, or, xor, and then, or else
=, /=, <, <=, >, >=, in, not in
+, -, &                 (kétoperandusú)
+, -                    (egyoperandusú)
*, /, mod, rem
**, abs, not
```

Így a következő kifejezések jelentése:

```
A / B * C               jelentése     (A / B) * C
A + B * C + D           jelentése     A + (B * C) + D
A * B + C * D           jelentése     (A * B) + (C * D)
A * B ** C              jelentése     A * (B ** C)
```

Ha azonos precedenciájú operátorokat alkalmazunk, akkor a kiértékelés balról jobbra történik és nem szükséges zárójeleket használni. A nyelv nem engedi meg a hatványozás operátor többszörös alkalmazását zárójelek nélkül. Tehát nem írhatjuk azt, hogy

```
A ** B ** C
```

hanem azt kell írnunk, hogy

```
(A ** B) ** C        vagy      A ** (B ** C)
```

Viszont megengedettek a következők:

```
A - B - C,          illetve    A / B / C
```

Figyeljünk oda az egyoperandusú - operátor precedenciájára:

```
-A ** B    jelentése   -(A ** B)     nem pedig  (-A) ** B
```

Hasonlóan

```
A ** -B          és         A * -B
```

helytelen, itt is zárójeleket kell használnunk.

Az abs precedenciája nem ugyanaz, mint az egyoperandusú - precedenciája. Így írhatjuk azt, hogy

```
-abs X
```

de azt nem, hogy

```
abs -X
```

ez utóbbit zárójelezve kell írnunk:

```
abs(-X)
```

3.4. A tömb típus

A tömb egy olyan összetett objektum, aminek komponensei ugyanazon altípushoz tartoznak – ez a tömb elemtípusa. Egy tömbkomponensre indexével, ill. indexeivel hivatkozhatunk, az indexértékek tetszőleges diszkrét típus elemei lehetnek. Minden tömbérték az elemtípusból vett értékek sorozata, amiket az indextípus(ok)ból vett értékekkel folyamatosan sorszámozunk. Az Adában a tömbök elemei bármilyen típusúak lehetnek. Egy tömb típust *korlátozott*nak nevezünk, ha a komponens típusa korlátozott.

```
type Napok is (Hétfô, Kedd, Szerda, Csütörtök, Péntek,
               Szombat, Vasárnap);
Munka: array(Napok) of  Natural;
```

Ekkor a `Munka` tömb természetes számok egy sorozata, ahol az egyes komponenseket a megfelelő indexeléssel érhetjük el: `Munka(Hétfô)` a tömb első eleme, `Munka(Kedd)` a második stb. Természetesen szűkíthetjük a lehetséges értékek halmazát a `0` és `24` közötti értékekre, valamint a lehetséges indexek halmazát is a `Munka(Hétfô)` és `Péntek` közöttiekre, így írhatjuk például:

```
subtype Munkaórák is Natural range 0..24;
Munka1: array(Napok range Hétfô..Péntek) of Munkaórák;
```

A `Munka1` tömb indexei már csak `Hétfô` és `Péntek` közötti értékeket vehetnek fel, és a tömb egyes komponensei is csak `0` és `24` közötti egész értékek lehetnek. Ha mást írunk, akkor a fordító figyelmeztet, ha felismeri, vagyis, ha ez a programszövegből egyértelműen kiderül. A `Munka1` tömböt használva írhatjuk például:

```
Munka1(Hétfô):=12;
Munka1(Kedd):=8;
Munka1(Szerda):=5;
Munka1(Csütörtök):=11;
Munka1(Péntek):=15;
-- de hibajelzést kapunk a következökre:
Munka1(Szombat):=6;  -- vagy:
Munka1(Hétfô):= 44;
```

Majd ebből kiszámíthatjuk a heti összmunkaórát:

```
összmunka := 0;
for I in Hétfô..Péntek loop
  összmunka := összmunka + Munka(I);
end loop;
```

Ha a hiba fordítás közben nem dönthető el, akkor futás közben kapunk a fenti összegzés után `Constraint_Error` hibajelzést, ha például a következőt írjuk: `Munka1(Hétfô):=összmunka;`

Vizsgáljuk meg, hogy mi jellemez egy tömböt! Az első és legfontosabb a komponensek típusa. Ezen kívül meg kell adni indexeinek számát és típusát – az indexek sorrendje szignifikáns (amikor egy tömb egy komponensére hivatkozunk, általában nem mindegy, hogy például az I. sor J. eleméről van-e szó, vagy a J. sor I. eleméről).

Az Ada - számos más programozási nyelvtől eltérően – a tömb típusok definiálásánál meg is elégszik ennyivel. A konkrét indexhatárokat elegendő a tömb objektum definiálásakor meghatározni. Mit jelent ez a gyakorlatban?

A tömb típus definiálásának két formája van: a rögzített és a megszorítás nélküli indexhatárral való megadás. Az első – talán úgy is mondhatjuk, hogy a „hagyományos" – esetben a tömb típus definiálásakor rögzítjük a pontos méretét is. Például a

```
type Táblázat is array (Integer range 1 .. 6) of Integer;
```

definíció esetén a Táblázat típusú tömbváltozók indexei egységesen 1 és 6 közötti egész értékek lehetnek. Ez a megszorítás jelentősen korlátozza a típust használó alprogramok újrafelhasználhatóságát, hiszen ekkor, ha például megvalósítunk egy rendezést vagy egy keresést a `Táblázat` típusú elemeken, akkor, ha legközelebb 5 vagy 7 hosszúságú vektorokat szeretnénk rendezni, új típust kell bevezetnünk például a következőt:

```
type Táblázat7 is array(Integer range 1..7) of Integer;
```

és módosítanunk kell az alprogram specifikációját is, sőt, ha a törzsben hivatkoztunk a konkrét indexhatárokra, akkor a törzset is, vagy, – ha az előző változatra is szükség van még –, akkor egy (vagy több) módosított másolatát kell létrehozni. Ez a tevékenység, amellett, hogy a fejlesztő számára többletmunkát jelent, s így csökkenti munkája hatékonyságát, rontja a program áttekinthetőségét, megbízhatóságát és hatékonyságát is.

A megszorítás nélküli indexhatárok használatával azonos típusúnak tekinthetjük mindazokat a tömböket, amelyek egy adott típusú elemeket tartalmaznak, és indexeik ugyanolyan típusúak, bár az indexhatáraik különbözőek. A következő

```
type Vektor is array (Integer range <> ) of  Integer;
--vagy
type Mátrix is array (Integer range <>, Integer range <>)
    of Float;
```

jellegű típusdefiníciók használatával a változó deklarálásakor kell csak eldönteni a konkrét méretet, és például a `V1, V2, V3` változók azonos típusúak lesznek:

```
V1: Vektor (1..3);
V2: Vektor (2..4);
V3: Vektor (1..8);
```

Hasonlóan az `A` és az `Inverz`, mindkettő `Mátrix` típusú:

```
A : Mátrix(1 .. 2, 1 .. 5);
Inverz : Mátrix (1 .. N, 1 .. N);
```

Az Ada a definíciókat dinamikusan értékeli ki, így az előző példában `N` lehet változó, sőt akár függvényhívás is.

A határozatlan indexhatárú tömbtípusokból indexmegszorítás megadásával képezhetünk olyan altípust is, amely már alkalmas objektumok definiálására. Így például

```
type Beosztás is array(Napok range <>) of Boolean;
subtype Munkanap is Beosztás(Hétfô..Péntek);
```

esetén, ha `Beosztás` típusú objektumot (változót vagy konstanst) akarunk definiálni, kötelező az indexhatárok megadása is:

```
Kati_Beosztása : Beosztás(Kedd.. Szombat);
```

ugyanakkor, a `Munkanap` típusú objektumokra a már megadott határok használhatóak.

```
Imi_Beosztása : Munkanap;
```

Meghatározatlan méretű tömb típus lehet alprogram formális paraméterének típusa is,

```
procedure Add_Hozzá(A : in Vektor; B : in out Vektor);
```

ilyenkor az alprogram meghívásakor az aktuális paraméter definiálja majd a konkrét indexhatárokat.

A tömbök elemeire a szokásos módon, indexeléssel hivatkozhatunk, az index(ek)et `()` közé írjuk. Például: `Imi_Beosztása(Hétfô)`, vagy az előző `Inverz` mátrixnál az `Inverz(I,J)` érvényes hivatkozások. Vektorokból képzett vektor esetén – például

```
type Sor is array(1..Max_Sor_Méret) of  Character;
type Lapok is array(Integer range <>) of  Sor;

Lap:Lapok(1..55);
```

definíciók után írhatunk Lap(I)(J)-t. A tömbök elemei *nem lehetnek* meghatározatlan méretűek! Ha érvényes a

```
function Köv_Mátr(K : Positive) return Mátrix;
```

függvénydefiníció, akkor a Köv_Mátr(L)(I,J) is szabályos indexelés, azaz nem csak a változók, hanem a tömb típusú objektumot visszaadó függvényhivatkozások is indexelhetőek. Megengedett, hogy az egydimenziós tömbök egy részére (szeletére) hivatkozzunk, például V1(Tól..Ig) formában. (Megjegyezzük, hogy V1(1..1) és V1(1) nem egyenértékű, hiszen az első esetben a V1 egyelemű résztömbjét írtuk – ez a kifejezés Vektor típusú –, a másodikban a V1 első elemét, – ez Integer típusú!)

Egy tömb objektum deklarálásánál megadhatunk közvetlenül egy (névtelen) tömb típus definíciót is – ekkor természetesen meg kell adni valamilyen formában az indexmegszorítást is. Például:

```
X : array(1..80,1..100) of Boolean;
Vegyes : array (Napok range Hétfô..Szerda) of Boolean;
Lap1 : array (Positive range <>) of Sor :=
   (1|50 => Sor'(1|Sor'Last => '+', others=>'-'),
    2..49 => Sor'(1|Sor'Last => '|',  others=>' ');
-- itt a Lap1-re a kezdeti értéke adja a megszorítást
```

Egy tömbtípus minden kiértékelése egy új, az eddigiektől különböző tömb típust hoz létre, így minden olyan objektum, amelynek a deklarációjában szerepel egy tömbtípus-definíció, más-más típusú lesz.

3.4.1. Tömb típusú objektumok műveletei

Korábban említettük, hogy a korlátozás nélküli tömbtípusok alprogramok formális paraméterének típusaként használhatóak, és hogy ekkor majd az aktuális paraméter mérete határozza meg a konkrét indexhatárokat. Az alprogram törzsében azonban általában valamilyen módon hivatkozni kell a formális paraméter lehetséges indexeire. Többek között ezt teszik lehetővé a tömb típusú objektumokra definiált attribútumok: (legyen X egy tetszőleges K-dimenziós tömb, N pedig egy statikus kifejezés):

X'First – jelöli az X tömb első index- intervallumának alsó határát, típusa a megfelelő index típus.

X'First(N) – jelöli az X tömb N. index-intervallumának alsó határát, típusa a megfelelő index típus.

X'Last – jelöli az X tömb első index-intervallumának felső határát, típusa a megfelelő index típus.

`X'Last(N)` – jelöli az `X` tömb `N`. index-intervallumának felső határát, típusa a megfelelő index típus.

`X'Range` – jelöli az `X` tömb első index-intervallumát, vagyis ekvivalens az `X'First..X'Last` által meghatározott érték-intervallummal.

`X'Range(N)` – jelöli az `X` tömb `N`. index-intervallumát, vagyis ekvivalens az `X'First(N)..X'Last(N)` által meghatározott érték-intervallummal.

`X'Length` – jelöli az `X` tömb első index-intervallumában lévő elemek számát, típusa *universal_integer*.

`X'Length(N)` – jelöli az `X` tömb `N`. index-intervallumában lévő elemek számát, típusa *universal_integer*.

Általában lehetséges, hogy egy implementáció a többdimenziós tömböket sorfolytonosan ábrázolja, a többdimenziós tömb-aggregátumokra használt jelöléssel összhangban. (Ha a `Convention(Fortran,...)` pragmát (B.5. Függelék) alkalmazzuk egy többdimenziós tömbre, akkor oszlopfolytonos lesz a sorrend.) Az attribútumok segítségével már nagyon egyszerűen megírhatjuk például az előbb specifikált `Add_Hozzá` eljárás törzsét:

```
procedure Add_Hozzá(A: in Vektor; B: in out Vektor) is
  begin  -- Feltételezzük, hogy A'Range = B'Range.
    for I  in A'Range  loop
      B(I) := B(I) + A(I);
    end loop ;
end Add_Hozzá;
```

(Attribútumok használatával kellene ellenőrizni természetesen, hogy az aktuális paraméterek egyáltalán összeadhatóak-e.)

Ha a tömb nem korlátozott, akkor definiált az értékadás művelete, egy tömböt (vagy résztömböt) értékül adhatunk egy másik, ugyanilyen típusú és méretű tömbnek, így megengedettek a `V2 := V1;` vagy `V3(3..5) := V1;` jellegű értékadások.

Azonos típusú nem korlátozott tömbökre alkalmazható a = és a /= reláció, továbbá diszkrét értékű vektorokra a <, <=, >, >= (lexikografikusan értelmezett) relációk.

Az azonos hosszúságú logikai értékeket tartalmazó vektorokra értelmezték az `and`, `or`, `xor`, és a `not` műveleteket, amelyek eredménye olyan logikai vektor, amelynek indexhatárait a baloldali argumentum indexhatárai adják, és elemenként tartalmazza a megfelelő logikai művelet eredményét.

A korlátozatlan indexhatárú egydimenziós tömb típusokra értelmezték a konkatenáció (`&`) műveletét is, a konkatenáció egyik, vagy akár mindkét operandusa lehet a tömb típus komponens típusának értéke is. Az eredmény

alsó határa ekkor a baloldali operandus alsó határa, ha az tömb, és az index-típus alsó határa, ha a baloldali operandus a tömb egy komponens értéke. Így például, a

```
type Konk is array(Napok range <>) of Integer;
```

tömbtípus esetén deklarálhatjuk a következő változókat:

```
    K1: Konk(Kedd..Szerda);
    K2: Konk(Szerda..Csütörtök);
    K3: Konk:=K1&K2;
    K4: Konk:=K1&1;
    K5:  Konk:=2&3;
    K6: Konk:=1&K1;
```

Ekkor `K3'First` és `K4'First` értéke `Kedd` lesz (`K1'First`); `K3'Last` értéke `Péntek` és `K4'Last` értéke `Csütörtök`, de `K5'First` és `K6'First` értéke `Hétfô` – vagyis `Napok'First`.

Az Ada lehetőséget ad arra, hogy a tömb objektumoknak ne csak komponensenként adhassunk értéket, hanem szükség esetén feltölthessük az egész tömböt egyszerre is. Ez az ún. *tömb-aggregátum*ok segítségével valósítható meg, ekkor a tömböt vagy *„pozíciós" formában*, az elemek előfordulási sorrendjében adhatjuk meg, például a

```
Munka1 := (7,6,12,14,3);
```

utasítással a `Munka1(Hétfô)` értéke `7` lesz, a `Munka1(Kedd)` értéke `6` stb., vagy *„név szerinti" formában*, (itt az egyes komponensek „neve" az indexe), például a

```
Munka1 := ( Hétfô => 7, Kedd => 6, Szerda => 12,
   Csütörtök => 14, Péntek => 3);
```

alakban. Név szerinti forma esetén az értékeket nem kötelező az indexek növekvő sorrendjében megadni. Az azonos értékek összevonhatók, ekkor `'..'` használható az index-intervallumok, — a vagylagos választékok leírására, például a

```
Munka1 := (Hétfô..Kedd=>9, Szerda | Péntek=>12, Csütörtök=>3);
```

utasítás a `Munka1` tömb `Hétfô`-től `Kedd`-ig terjedő indexű értékeit `9`-re, a `Szerda` és a `Péntek` indexűeket `12`-re, míg a `Csütörtök` indexűt `3`-ra állítja be. Utolsó lehetőségként írhatunk `others`-et is, ez a fennmaradó indexek értékeit állítja be. Például:

```
Munka := (Hétfô..Péntek=>8, others=>0);
```

A tömb-aggregátumok a tömbök inicializálására is használhatók, és ennek segítségével adhatunk meg tömbkonstansokat is a deklarációs részben:

```
Köt_Óra: constant array (Napok range Hétfô..Péntek)
                              of Munkaórák := (others=>8);
```

Az aggregátumok többdimenziós tömbökre is használhatóak, egy `N` dimenziós tömböt úgy tekinthetünk, mint egy olyan `1` dimenziósat, amelynek az elemei `N-1` dimenziósak:

```
M: Mátrix(1..2, 1..3):=
            (1 = > (1.1,1.2,1.3), 2 = > (2.1,2.2,2.3));
```

Megszorítás nélküli tömb típusok esetén aggregátumok segítségével definiálhatjuk az indexhatárokat is, ezeket ui. akkor nem kötelező a deklarációban előírni, ha a kezdeti érték egyértelművé teszi. Például:

```
Cm: Mátrix := (1 .. 5 = > (1 .. 8 = > 0.0));
```

A `Cm'First = 1`, `Cm'Last = 5`, `Cm'First(2) = 1` és `Cm'Last(2) = 8` lesz ekkor. (Érdemes megjegyezni, hogy ha az aggregátumban pozíció szerinti formát adunk meg egy korlátozás nélküli tömbtípus esetén, akkor a kapott objektum indexének alsó határa az indextípus alsó határa lesz.[3]) Az `others` nem használható akkor, ha az aggregátum indexhatárait a környezete nem határozza meg egyértelműen. Például:

```
   if Cv < ( others = > 2) then ... --  hibás!
```

3.4.2. Szöveg típusok

A karakter típusú komponenseket tartalmazó egydimenziós tömb típusokat hívják szöveg típusoknak. Két előre definiált szöveg típus van, a `String` és a `Wide_String`, mindkettő a predefinit `Positive` típussal indexelve, a `Standard` csomag látható részében:

```
subtype Positive is Integer range 1..Integer'Last;
type String is array(Positive range <>)
                                           of Character;
type Wide_String is array(Positive range <>)
                                           of Wide_Character;
```

[3] Így például `Cv: constant Vektor := (1,2,3,4,5);` definíció után a `Cv'First = -2147483648` lesz GNAT Ada 95 fordítóját használva.

Helyesek a következő típusdefiníciók és deklarációk:

```
Csillagok: String(1..120) := (1..120=>'*');
Vmi: constant String := "Mi ez?";
Ket_Vmi: String := Vmi & Vmi;

type Római_Számjegy is ('I','V','X','L','C','D','M');
type Római_Szám is array(Positive range <> )
                                of Római_Számjegy;
Kilencvenhat:  constant Római_Szám := "XCVI";
S1:  constant String := "GIN";
S2:  constant String := ('G','I','N');
Null_String:  constant String(1..0):="";
```

Végül készítünk egy eljárást a `String` típus használatának bemutatására, amely megfordít egy szöveget:

3.1. Példa

```
procedure Stringet_Fordít (S: in out String) is
   L: constant Natural := S'First;
   U: constant Natural := S'Last;
   C: Character;
begin
   for I  in L .. (L + U) / 2  loop
      C := S(I); S(I) := S(U - I + 1); S(U - I + 1) := C;
   end loop;
end Stringet_Fordít;
```

3.5. A rekord típus

A rekord típus egy olyan összetett típus, amelynek komponens típusai különböző, már deklarált típusokhoz tartozhatnak. A rekord típus az Adában a direkt szorzat és az unió típuskonstrukciók megvalósítását teszi lehetővé. A komponensek nevei különbözőek kell legyenek (ezek ugyanabban a deklarációs egységben vannak). A rekord típusú objektumokban a komponensekre névvel, ún. szelektorral hivatkozhatunk.

Az Ada megengedi, hogy a rekord típus definíciójában megadjuk a típus objektumainak kezdőértékeit.[4]

A rekord típus előre definiált műveletei a minősítés – a rekord típusú objektumok komponenseire minősítéssel hivatkozhatunk –, a tartalmazásellenőrzés („membership test”) és az explicit konverzió. Ha a rekord nem korlátozott, akkor az értékadás és az egyenlőség vizsgálat is megengedett.

[4] Csak akkor nem használhatunk kezdőértéket, ha a rekord komponens korlátozott.

3.2. Példa

```
type Hónap is (Január, Február, Március, Április,
               Május, Június, Július, Augusztus,
               Szeptember, Október, November, December);

type Dátum is
  record
    Év: Integer;
    Hó: Hónap;
    Nap: Integer range 1 .. 31;
end record ;

type Complex is
 record
    Valós, Képzetes: Float := 0.0;
 end  record ;

D: Dátum;
C1: Complex;
C2: Complex ;
```

A `C1` és `C2` `Complex` változók valós és képzetes része a típusra vonatkozó kezdeti értékadásnak megfelelően `0.0` lesz.

A rekord mezőire a szokásos szelekciós jelöléssel hivatkozhatunk, tehát például a `D` `Dátum` típusú változó `Év`, `Hó` és `Nap` mezőire `D.Év`, `D.Hó`, `D.Nap` formában.

A rekord típus definíciójában szerepelhet a `limited` kulcsszó – az ilyen típusú rekord objektumokat *korlátozott* típusúaknak hívjuk, s ezekre nem megengedett az értékadás és az egyenlőségvizsgálat. Megengedett természetesen a komponensekre vonatkozó értékadás, ha a komponensek nem korlátozott típusúak.

3.3. Példa

```
type Arek is limited record
  X: Integer:=0; -- megengedett a kezdeti értékadás,
             -- mert az Integer nem korlátozott típus
end record;
 A, B: Arek;
....
   A.X := 1;   --megengedett
   A := (2);   -- aggregátummal nem adhatunk értéket
               -- a rekord egészének
   B := A;     -- nem adhatjuk értékül a teljes rekordot
   B.X := A.X; -- a komponensek között megengedett
```

```
              -- az értékadás
  if A=B then ..... end if; -- nem megengedett a teljes
                        -- rekordok összehasonlítása
```

A tömb típusokhoz hasonlóan a rekord típusoknál is lehetséges egy aggregátummal az egész rekordnak egyszerre értéket adni. [5] Felsorolhatjuk a mezők sorrendjében azok értékeit – ez az ún. *pozíció szerinti forma* –, például:

```
D := (1776, július, 4);
```

vagy felírhatjuk a szelektorokat, mindegyik mellé odaírva a mező értékét (ez az ún. *név szerinti forma*), például:

```
D := (Nap => 4, Hó => Június, Év => 1998);
```

Az utóbbi esetben a mezők felsorolásának sorrendje közömbös. A két formát keverhetjük is, például írhatjuk:

```
D := (4, Év => 1998, Hó => Június);
```

de ekkor a pozíció szerinti rész meg kell előzze a név szerinti hivatkozás(oka)t. Ha egy aggregát típusa nem egyértelmű, akkor a típus nevével ún. minősítést használhatunk, például: `Complex'(0.0,1.0)`.

Rekord típust használhatunk például változó hosszúságú szövegek tárolására is, amelynek egyik komponensében a lehetséges maximális hosszú szövegnek foglalunk helyet, s másik komponense mutatja a benne tárolt szöveg aktuális hosszát, ez kezdetben `0`, s később megfelelően változtatható:

```
Max_Méret: constant Integer :=5;
type Név is
  record
    Hossz: Integer range 0 .. Max_Méret := 0;
    Érték: String(1 .. Max_Méret);
end record ;
```

Az `N1: Név;` deklaráció után az `N1.Hossz` komponens értéke `0` lesz. (Ez a megoldás csak akkor megbízható, ha a `Max_Méret` konstans, különben előfordulhat, hogy egy belső blokkban megváltoztatjuk az értékét, s csak futás közben jelez hibát.)

[5] Ha a rekord típus nem korlátozott.

3.5.1. A rekord diszkriminánsai

A nem tömb összetett típusoknak lehetnek paramétereik, ezt az Adában *diszkriminánsnak* hívják. Egy objektum diszkriminánsa az objektum egy komponense, és vagy *diszkrét*, vagy *mutató* típusú (ez utóbbi csak a taszk, a védett, és a korlátozott típusoknál megengedett).

A rekord típusoknál a diszkriminánsok segítségével leírhatjuk például a rekord egy tömb komponensének az indexhatárát. Például:

```
type Mátrix_rek(Sorok, Oszlopok: Integer) is
   record
      Mat: Mátrix(1..Sorok,1..Oszlopok) ;
   end record ;
```

A rekord diszkriminánsának lehet alapértelmezett kezdeti értéke.[6] Például:

```
subtype Szöveg_Méret is Integer range 0..1000;
type Szöveg(Méret: Szöveg_Méret:=100) is
   record
      Pozíció: Szöveg_Méret :=0;
      Érték: String(1..Méret):=(1..Méret=>' ');
   end record ;
```

Hasonlóan a határozatlan indexhatárú tömbhöz, a kezdeti érték nélküli paraméteres rekord típus sem alkalmas közvetlenül objektum (változó vagy konstans) definiálására. Így nem megengedett például az `M: Mátrix_rek;` deklaráció, csak a diszkrimináns értéket tartalmazó `M: Mátrix_rek(N,L);` alakú.

Szerepeltethető viszont az ilyen típus alprogram formális paramétereként, és ilyenkor az alprogram meghívásakor az aktuális objektum definiálja a diszkrimináns értékét.

Abban az esetben, ha a rekord diszkriminánsai kezdeti értékkel rendelkeznek, a típus közvetlenül is használható objektum-definiálásra, ekkor a diszkrimináns egyszerűen felveszi a kezdeti értékét, a fordítóprogram pedig a legnagyobb változathoz tartozó memóriát foglalja le, így a diszkrimináns értéke később – *az egész rekordra vonatkozó értékadással* – változtatható.

Például a `Szó: Szöveg;` változó deklarálásakor a `Szó` változó `Méret` paramétere `100` lesz, és a `Szó := (3,2,"IMI");` utasítással a `Méret` paramétert `3`-ra változtathatjuk. Kötelező ekkor az egész rekordnak értéket adni, hiszen ha a diszkrimináns például a rekord típus egy tömb típusú komponensének volt az indexhatára, akkor az így kezelt rész hossza is változik.

[6] Feltéve, hogy nem jelölt típusnál, és nem minta formális paramétereként szerepel.

Az objektumok definiálásakor a paraméterek aktuális értékeivel „megszorítjuk" a típust, (ez kötelező, ha nincs kezdeti érték, de lehetséges akkor is, ha van), s az eredeti rekord típus egy-egy altípusát hozzuk létre. A `T: Szöveg(5);` és az `R: Szöveg(10);` változók így különböző altípusba esnek, és például a `T:=R;` értékadás hibás lesz, hiszen eltérő méretű `Érték` komponensnek foglaltunk helyet.

Fontos szabály tehát, hogy ha a típus felhasználásakor az objektum definiálásakor explicit módon megadunk a diszkriminánshoz egy megszorítást, ezzel a típus egy altípusát állítjuk elő. Ha viszont a diszkrimináns kezdeti értékét használjuk fel, azzal nem szűkítjük le a típust.

A kezdeti értéket is tartalmazó paraméterezett rekord típusok esetén, ha ez a típus egy alprogram formális paramétere, akkor az `A` aktuális paraméterről az `A'Constrained` attribútummal lekérdezhető, hogy az `A` megszorított érték-e vagy sem. `A'Constrained` a `True` értéket adja vissza, ha `A` egy konstans, egy érték, vagy egy „megszorított" változó.

Az előző `Név` típus diszkriminánssal ellátott rekordokkal általánosabban leírható:

```
subtype Méret is Integer range 0..50;
subtype Név2(Max_Méret: Méret)  is
   record
      Hossz: Méret := 0;
      Érték: String(1 .. Max_Méret);
   end  record ;
```

A `Név2` típus második komponensének értéke itt olyan szöveg lehet, aminek maximális hossza `Max_Méret`. A `Hossz` komponenst arra használhatjuk, hogy jelezzük az `Érték` vektorban elhelyezett karaktersorozat valódi hosszát.

```
 N: Név2(9);
 N := (9,3,"IMI      ");  --bár a Max_Méret diszkrimináns
  --nem változtatható, az aggregátumba le kell írni!
 M: Név2;  -- csak akkor lenne írható, ha a
           -- diszkriminánsnak kezdeti értéke van!
```

Felmerül a kérdés, nem lehetne-e a diszkriminánsok segítségével egy rugalmas szöveg típust készíteni, például:

3.4. Példa

```
...
type Rugalmas_Szöveg(Max: Integer := 0) is
  record
    Karakterek: String(1..Max) := (1..Max=>' ');
  end record;
```

```
X: Rugalmas_Szöveg;
begin.....
  X := (10, (others => 'a')); --itt 10 hosszúnak tekinteni,
  X := (20000, (others => 'b')); -- s később 20000 hosszúnak
end;
```

A fordító ekkor a kezdeti értékkel megadott diszkriminánsokra vonatkozó szabály szerint a diszkrimináns típusának megfelelő maximálisan szükséges memóriát kellene lefoglalja, és az `Integer` típus típusértékhalmaza ehhez túl nagy. Így már a fordító figyelmeztet, hogy futás közben `Constraint_Error` hibajelzés várható.

A származtatott típusban a diszkrimináns lehet az ős típus paraméterének egy aktuális értéke:

3.5. Példa

```
type Négyzet(Oldal: Integer) is
    new Mátrix_rek(Sorok => Oldal, Oszlopok => Oldal);
type Dupla_Négyzet(Szám: Integer) is
  record
    Bal: Négyzet(Szám);
    Jobb: Négyzet(Szám);
  end record ;
```

3.5.2. A variáns rekord

Az unió típus megvalósítását sok programozási nyelv a rekord típus segítségével teszi lehetővé. Az Adában diszkrimináns felhasználásával alakítható ki az ún. variáns részt tartalmazó rekord, ami unió típus definiálására használható – így két (vagy több) -féle adattípust fizikailag ugyanazon a memóriaterületen tárolunk. A nyelv – eltérően például a Pascaltól, vagy a C-től – biztonságosan kezeli ezt a típuskonstrukciót. Az, hogy egy objektum aktuálisan melyik bázistípusba tartozik, a rekord paraméterében rögzítve van, s ezt a mezőt csak az egész rekordra vonatkozó értékadással (aggregátummal) lehet megváltoztatni.

A variáns részt kiválasztó diszkriminánst a rekord típus belsejében a `case` kulcsszó után írjuk, majd a `when` kulcsszó után kell felsorolni – statikus kifejezésekkel – az unió típus különböző eseteit, itt használhatunk nem üres intervallumot (például `Hétfô..Szerda`), illetve a | jelet a „vagy" kifejezésére is. Az esetmegjelölések között minden lehetséges értéket fel kell sorolni (vagy pedig – utolsóként – az `others` lehetőséget kell használni). Kétszer nem szerepelhet ugyanaz! Ezután a => jelet követően kell megadni a megfelelő típusszerkezeteket. A variáns részt az `end case` zárja.

Például mondhatjuk, hogy az `Emberek` halmaza a `Házas`, az `Özvegy`, az `Egyedülálló` és az `Elvált` családi állapotú személyek uniója, s a fenti

esetek szerint az alapinformációkon (név, nem, születési dátum, gyermekek száma) kívül tárolni szeretnénk házasoknál a házastárs nevét, özvegyeknél a házastárs halálának dátumát, elváltaknál a válás dátumát is. Ekkor létre kell hoznunk egy felsorolási típust, amelyikkel meg tudjuk különböztetni az unió típus eseteit, majd az `Ember` típust például a következő rekord szerkezettel írhatjuk le:

3.6. Példa

```
type Állapot is
    (Egyedülálló, Házas, Özvegy, Elvált);
subtype Név is String(1..25);
type Nem is (Nô, Férfi);
type Ember (Családi_Áll: Állapot := Egyedülálló) is
  record
    Neve: Név;
    Neme: Nem;
    Születési_Dátum: Dátum;
    Gyermekek_Száma: Natural;
    case Családi_Áll is
      when Házas => Házastárs_Neve: Név;
      when Elvált => Válás_Dátuma: Dátum;
      when Egyedülálló =>  null ;
      when Özvegy => Halál_Dátuma: Dátum;
     end case ;
end record ;
```

Ekkor deklarálhatunk például egy `István: Ember;` típusú változót, s ez alapértelmezésben `Egyedülálló` lesz, és ez a teljes rekordra vonatkozó értékadással változtatható, így írhatjuk:

```
István := (Házas, ....);
```

Ugyanúgy, ahogy a diszkrét típusoknál az értékkészlet megszorítását használjuk altípusképzésre, a variáns rész egy variánsának megkötése szolgál az unió rekord típus megszorítására. Például az `Ember` típusból altípust is definiálhatunk:

```
subtype Egyedülállók is Ember(Egyedülálló);
Kati: Egyedülállók;
```

vagy közvetlenül egy, az altípusba eső változót:

```
Kálmán: Ember(Egyedülálló);
```

s ezeknél a változóknál a diszkrimináns már nem változtatható.

Altípusképzéskor az összes diszkriminánst meg kell adni (vagy a pozíciós, vagy a névvel jelölt formában).

Ellenőrizhetjük, hogy egy változó aktuálisan egy altípushoz tartozik-e, például

```
Imi:Ember;
```

deklaráció után:

```
if Imi in Egyedülállók then ... end if;
```

Származtatott típusok esetén az explicit típuskonverzió is megengedett:

```
....
type Valaki is new Ember;
X: Valaki;
...
X := Valaki(Imi);
```

Hasonlóképpen a `Személy` típusú objektumok például lehetnek `Tanárok` vagy `Diákok`, s ettől függően tárolhatjuk az általuk oktatott hallgatók létszámát, vagy azt, hogy nappali, illetve esti képzésben vesznek-e részt, s mi az utolsó átlaguk.

3.7. Példa

```
type Magasság is range 0..200;
subtype Név_Hossz is Natural range 0..20;
subtype Átlag is Float range 0.0..5.0;
type Foglalkozás is (Tanár, Diák);
type Személy(Kar_Szám: Név_Hossz := 0;
             Fogl: Foglalkozás := Diák) is
  record
    Név: String(1..Kar_Szám);   -- a név,
    case Fogl is                -- variáns rekord
      when Tanár => Oktatottak_Száma: Natural;
      when Diák => Nappali: Boolean := True;
                   Akt_Átlag: Átlag;
    end case;
end record;
```

Ebben a példában láthatjuk, hogy egy rekord típusnak több diszkriminánsa is lehet, de ha kezdeti értéket akarunk adni, akkor minden diszkriminánsának kell adni.

Ha egy `Személy` típusú `X` objektumban a `Fogl` diszkrimináns értéke `Tanár`, akkor a `Diák` variánshoz tartozó komponensekre történő bármely hivatkozás, például

```
X.Akt_Átlag := 4.5;
```

értékadás *futás közbeni hibajelzést* vált ki.

A variáns rekord diszkriminánsa használható például `case` utasításban az elágazás feltételeként:

3.8. Példa

```
type Alakzat is (Négyszög, Kör, Pont);
type Pozíció is
 record
    X, Y: FLOAT;
 end  record ;
type Geometriai_Alak (S: Alakzat) is
 record
  Kerület: Float;
  case S is
    when Négyszög = > A, B: Natural;
    when Kör = > Sugár: Float := 1.0;
    when Pont = > P: Pozíció;
  end case ;
 end  record ;

 function Terület(F: Geometriai_Alak)  return Float is
 Pi:  constant := 3.1614;
 begin
  case F.S is
    when Négyszög = >  return Float(F.A * F.B);
    when Kör = >  return Pi * F.Sugár ** 2;
    when Pont = >  return 0.0;
  end case ;
 end Terület;
```

Összefoglalva, az Ada a diszkriminánsokkal szemben megköveteli a következőket:

- Típusdefinícióban a diszkrimináns csak indexmegszorításként, a variáns rész neveként, vagy aktuális paraméterként használható. Minden esetben önmagában kell állnia, nem szerepelhet kifejezésben. (Egy diszkrimináns nem használható egy skalár komponens megszorításaként.)
- A variáns rész neve nem kezelhető egészen ugyanúgy, mint a rekord egy komponense. Csak akkor kaphat új értéket, ha az egész rekord új értéket kap.

- Ha a diszkriminánssal rendelkező típus újabb diszkriminánssal rendelkező típust tartalmaz, akkor ennek a diszkriminánsa csak a külső típus diszkriminánsaitól függhet.

3.6. Mutató típusok

Egy programegység vagy egy blokk deklarációs részében definiált objektum az illető egység memóriarészében kap helyet. Az objektum létezésének ideje alatt ennek az objektumnak a mérete és a címe állandó.

Azokat a típusokat, amelyek objektumainak mérete vagy a címe létrehozása után is változhat, *dinamikus típusoknak* nevezzük. Az Ada a pointer típus elnevezés helyett az `access` típus elnevezést használja. Azt a típust, amilyen típusú objektumokat jelölnek a mutatók, a dinamikus típus *gyűjtőtípusának* nevezzük.

Kétféle mutató típus létezik: az objektumokat, illetve programokat jelölő mutató típus. Mindkettő rendelkezik a `P'Access` attribútummal. Ha `P` objektum, akkor a `P`-vel jelölt objektum mutatóját adja vissza, ha `P` alprogram, akkor a `P` nevű alprogram memóriabeli címét adja vissza.

Minden mutató típusnak van egy speciális értéke, amelyet `null`-nak nevezünk, és ez azt jelenti, hogy a mutató nem vonatkozik semmilyen objektumra vagy alprogramra. Minden mutató típusú változó `null` értéket kap addig, amíg nem inicializáljuk egy konkrét értékkel. Az Adában nem végezhetünk összeadási és kivonási műveleteket a mutatókkal.

A nyelv kétféle objektumot különböztet meg: a deklarált objektumokat és a dinamikusan létrehozott objektumokat. A deklarált objektumokat egy programegység deklarációs részében hozzuk létre, míg a dinamikus objektumokat explicit allokálással. Az Ada95 megengedi, hogy mind a kétféle objektumra mutatót állítsunk.

Dinamikus objektum létrehozása `new` utasítással történik. Ha nincs elegendő memória, akkor a végrehajtás `Storage_Error` hibát vált ki.

Objektumra mutató típusok deklarálását a következőképpen végezhetjük:

```
type PInteger is access Integer;
```

`PInteger` típusú mutató csak dinamikusan létrehozott változókra mutathat. Ahhoz, hogy egy mutató egy programegység deklarációs részében deklarált változóira is és a dinamikusan létrehozott változókra is állítható legyen, az `all` kulcsszót használjuk:

```
type PInteger is access all Integer;
```

```
type PInteger is access constant Integer;
```

Ez utóbbi esetben a mutatót változókra is és konstansokra is lehet állítani. Ha egy deklarált objektumra mutatót akarunk állítani, akkor `aliased` típusúnak kell írjuk.

Ha egy `X` mutató típusú változó egy objektumra mutat, akkor az `X.all` a mutatott objektumot jelöli.

Konstansra állított mutatók esetén a következőkre kell figyelni:

```
type Konst_int is access constant Integer;
type Int is access all Integer;
P: Konst_int;
Q: Int;
I: aliased Integer;
K: aliased constant Integer := 20;

-- a következő értékadások helyesek:
P := K'Access;
P := I'Access;
Q := I'Access;
Q.all := 29;
P := new Integer'(0);

-- a következő értékadások pedig helytelenek:
P.all := 1; -- hibás értékadás, mert nem változtathatjuk meg
-- a P álltal jelölt értéket
Q := K'Access;  -- a Q változó értékekre mutat,
-- nem konstansokra
P := new Integer; -- helytelen, mert kell egy kezdeti érték
```

Nemcsak a mutatott objektum lehet konstans, hanem maga a mutató is.

```
type Konst_int is access constant Integer;
type Int is access all Integer;
I: aliased Integer;
K: aliased constant Integer := 20;
P: Konst_int;
Q: Int;
KP: constant Konst_int := K'Access;
KQ: constant Int := I'Access;
-- a következő értékadás helyes
KQ.all := 1;

-- a következő értékadások hibásak
-- konstans mutató értékét nem változtathatjuk meg
KP := new Integer'(100);
```

```
KQ := new Integer'(200);
KP.all := 10;  -- mivel KP konstans értékre mutat,
-- a mutatott érték sem módosítható
```

A `P` és `P.all` közötti különbséget a következő példa szemlélteti:

```
type P access Integer;
P1, P2: P;
...
P1.all := P2.all;
P1 := P2;
```

A `P1.all := P2.all;` értékadás után a `P1` által mutatott érték ugyanaz lesz, mint a `P2` által mutatott érték. A `P1 := P2;` értékadás után a `P1` és `P2` ugyanarra az objektumra mutat. Ha egy mutató értéke `null`, akkor az `X.all` semmire sem vonatkozik, így az `X.all` használata ilyenkor `Constraint_Error` hibát vált ki. A `.all` előtt állhat egy függvényhívás is:

```
V := Függvény(X, Y).all;
```

a `Függvény` visszaad egy mutatót, a mutató által jelölt érték pedig bemásolódik a `V` nevű változóba. Ha pedig fordítva írjuk:

```
Függvény(X, Y).all := V;
```

akkor a `Függvény` által visszaadott mutató olyan értékre mutat, ami megegyezik a `V` változó értékével.

Ha `Ptömb` egy tömbre állított mutató, akkor a `Ptömb.all(1)` a `Ptömb` által mutatott tömb első értékét jelöli, melyet úgy is rövidíthetünk, hogy `Ptömb(1)`.

Ugyanez érvényes a tömb szeletekre is: `Ptömb.all(2 .. 6)` helyett írhatunk `Ptömb(2 .. 6)`-t. Rövidítéseket attribútumokra is írhatunk, például: `Ptömb'Range(2)`. Rekordra állított mutató esetén `Prekord.all.K` helyett írhatjuk `Prekord.k`-t, ahol `K` a rekord egy mezője.

Ha `N` egy `Integer` és `A` pedig az `N` `Integer` értékre mutat, akkor `N` és `P.all` ugyanazt az objektumot jelölik. `P.all` az `N` változó *alias*-a.

```
type Tömb is array(1 .. 10) of Integer;
X: aliased Tömb;
```

Az `X` tömbre állított mutatót az `X'Access` attribútum segítségével hozhatjuk létre. Az `aliased` kulcsszó teszi lehetővé, hogy az `X` tömb címe lekérhető legyen.

Az `X(1)'Access` helytelen művelet, mivel a tömb egy elemére nem állíthatunk mutatót, csak az egész tömbre. Azonban ha a deklarálást a következőképpen írjuk:

```
type Új_Tömb is array of aliased Integer;
Y: Új_Tömb;
```

Ekkor `Y(1)'Access` megengedett és `Y` első komponensére állítja a mutatót. Azonban helytelen lesz az `Y'Access`, mert a tömb nem `aliased`, csak a komponense. A következő deklarálás esetén pedig

```
Z: aliased Új_Tömb;
```

`Z'Access` és `Z(1)'Access` is helyes, azonban a kettő nem azonos, mivel az egyik a tömb címét adja vissza, tehát az `Új_Tömb` típusra mutat, a másik pedig a tömb első elemének címét, azaz egy `Integer` típusra állított mutatót.

Egy névtelen típusra nem lehet mutatót állítani. Deklaráláskor írhatjuk ugyan, hogy:

```
V: aliased array (1 .. 20) of Integer;
```

azonban a `V'Access` értéket soha nem tudjuk használni, mivel a `V` változó névtelen típusára nem állíthatunk mutatót, mert a mutató deklarálásakor szükségünk lenne a `V` típusának a nevére.

A rekord komponensekre is állíthatunk mutatókat. Ehhez a rekord megfelelő mezőjét szintén `aliased`-nak kell deklarálni.

```
type R is record
A: aliased Integer;
B: Integer;
end record;

Ra: aliased R;
Rr: R;

Ra.A'Access;  -- helyes, mivel az A mezö aliased
Rr.A'Access;  -- helyes, mivel az A mezö aliased

Ra.B'Access; -- helytelen, mert a B mezö nem aliased
Rr.B'Access; -- helytelen, mert a B mezö nem aliased

Ra'Access; -- helyes, mert az Ra rekord aliased
Rr'Access; -- helytelen, mert az Rr rekord nem aliased
```

Dinamikusan létrehozott objektumok esetén a mutatott objektum kezdetben lehet meghatározatlan értékű:

```
Pint := new Integer;
```

de ugyanakkor allokáláskor meg is lehet adni a mutatott objektum értékét:

```
Pint := new Integer(0);
```

Ez első esetben `Pint` `Integer` típusú objektumra mutat, a második esetben pedig inicializálja is az objektumot 0-val. Ugyanez érvényes a tömbökre, tömbszeletekre, illetve rekordokra állított mutatók esetén.

```
S: String(1..3) := "Ada";
P := new String(1..3);
-- nincs kezdeti értéke a mutatott objektumnak
P := new String:= "Ada"; -- "Ada" a mutatott String
P := new String'(S);
-- S-ben található a mutatott objektum értéke
```

Két, ugyanazzal a típussal allokált és ugyanolyan kezdeti értékre állított mutató nem egyenlő, azaz:

```
P1, P2: PInteger := new Integer(0);
```

két különböző helyen létrehozott 0-ra mutat. Ha egy bizonyos helyen allokált 0-ra két mutatót is akarunk állítani, akkor előbb létrehozunk dinamikusan egy objektumot, amit 0-val inicializálunk, majd a két mutatót ugyanarra az objektumra állítjuk:

```
Pzero: constant PInteger := new Integer'(0);
P1, P2: PInteger := Pzero;
```

A mutató típus gyűjtőtípusa lehet diszkriminánssal ellátott rekord és határozatlan méretű tömb is. Ilyen esetben legkésőbb az objektum allokálásakor meg kell adni a megszorítást. Vagyis miután dinamikusan allokáltunk egy paraméteres objektumot, futás közben a méretét már nem változtathatjuk meg.

Az `X'Access` attribútum csak akkor használható, ha az `X` objektum legalább ugyanannyi ideig marad fenn, mint az attribútum által visszaadott mutató típusa.

3.9. Példa

```
procedure Program1 is
   type Integer_mutató is access all Integer;
   P: Integer_mutató;
   procedure Értékadás is
      I: aliased Integer := 0;
   begin
      P := I'Access;
```

```
         -- hibás értékadás, mert I az eljárás után megszűnik,
         -- míg a P típus nem szűnik meg
   end Értékadás;
begin
   Értékadás;
   P.all := P.all + 1;
      -- a P által mutatott érték nem létezik,
      -- az összeadás eredménye és ennek helye bizonytalan
end Program1;
```

Ha a mutató típust és egy ilyen típusú változót jóval magasabb szinten definiáltuk, és a mutatót egy belső blokkban deklarált lokális változóra állítjuk, akkor a blokk megszűnte után a mutató megmarad, azonban a lokális objektumot már nem érhetjük el.

A dinamikus objektumok addig maradnak fenn, amíg a típus definícióját tartalmazó programegység vagy blokk aktív.

3.10. Példa

```
procedure A is ...
   procedure B is ...
      type T is ...
      type PT is access T;
      G: PT;
      procedure C is ...
         L: PT;
      ...
      begin
         L := new T;
         ...
         G := L;
      end C; -- felszabadul L, a mutatott objektum,
             -- de nem szabadul fel a mutató típus
   ...
   end B; -- itt szabadul fel a mutató típus
          -- itt biztos, hogy az objektumra nem mutathat semmi
end A;
```

Egy másik példa muatatókkal:

3.11. Példa

```
procedure Program2 is
   type Integer_mutató is access all Integer;
   P: Integer_mutató;
   procedure Értékadás is
      type Lokális_mutató is access Integer;
```

```
      I: Lokális_mutató := new Integer'(0);
   begin
      P := I.all'Access;
      -- hibás értékadás, mert P típusa az eljárás után nem
      -- szűnik meg, míg az I típusa megszűnik az eljárás után
      end Értékadás;
begin
   Értékadás;
   P.all := P.all + 1;
      -- a P által mutatott érték nem létezik,
      -- az összeadás eredménye és ennek helye bizonytalan
end Program2;
```

A mutató típus élettartamára vonatkozó szabály érvényes alprogramot jelölő mutatók esetén is.

3.12. Példa

```
procedure Program3 is
   type Alprogram_mutató is access procedure;
   P: Alprogram_mutató;
   procedure Értékadás is
      I: Integer := 0;
      procedure Növel is
      begin
         I := I + 1;
      end Növel;
   begin
   P := Növel'Access; -- hibás, mert a Növel eljárás hamarabb
                      -- megszűnik, mint a P típusa
   end Értékadás;
begin
   Értékadás;
   P.all; -- a P által mutatott érték nem létezik,
end Program3;
```

Néha a mutató típus élettartamára vonatkozó szabály akadályozhat, ezért ennek elkerülésére használhatjuk az `'Unchecked_Access` attribútumot az `'Access` helyett. Ezt csak objektumokra lehet alkalmazni, alprogramokra nem. Ennek használata azonban nagyon veszélyes lehet, a fenti `Program1` és `Program2` az `'Unchecked_Access` attribútummal az `'Access` attribútum helyett már nem lesz helytelen, azonban a programok viselkedése nem határozható meg.

Az `'Unchecked_Access` attribútummal a fordító program nem ellenőrzi azt, hogy a mutatott objektum élettartama megegyezik-e a mutató típusá-

nak hatáskörével, a programozó az attribútumot a saját felelősségére használja.

Két különböző típusú mutató közötti konverzió a következőképpen valósul meg: ha `P` egy `T` típusú mutató és `P1` egy `T1` típusú mutató, akkor a `P := T(P1)` ekvivalens a `P := P1.all'Access` értékadással. `P` és `P1` objektumokra is és alprogramokra is mutathat. Ha `P1 = null`, akkor `P` is `null`.

Az Ada a biztonságosság és a hatékonyság miatt nem tartalmaz explicit felszabadító utasítást. A típus hatáskörének elhagyása után a rendszer felszabadítja a területet és újrahasználhatóvá teszi.

Egy mutató felszabadítása az `Ada.Unchecked_Deallocation` segítségével történik. A rendszer ezt hívja meg minden olyan mutató típusú objektum esetén, amely felszabadítható objektumra mutat. A programozó saját felelősségére használhatja ezt az eljárást:

```
procedure Free is new Ada.Unchecked_Deallocation(Integer,Mut);
```

Az `Ada.Unchecked_Deallocation`-nak két paramétere van, az első az objektum típusát, a második az objektumra állított mutató típusát jelöli. Ha a `P` egy `Mut` típusú változó, akkor a `Free(P)` eljárás meghívása után a mutatott objektum megszűnik, míg a `P` értéke `null` lesz. Nem használhatjuk ezt az eljárást deklarált objektumok esetén, csak a dinamikusan létrehozott objektumokra alkalmas.

```
P1 := new Integer;
P2 := P1; -- P1 és P2 ugyanarra az objektumra mutat
P1.all := 25; -- a mutatott érték 25
Free(P1); -- a P1 és P2 által mutatott objektum felszabadul
-- és P1 = null
Put(P2.all);
```

Ada95-ben bevezettek egy új típusú `in` paramétert, az `access` paramétert:

```
procedure P(A: access T);
```

ahol az aktuális paraméter egy olyan mutató lehet, amely ugyancsak `T` típusra mutat. Ugyanakkor nem lehet `null` az értéke, így nem kell külön tesztelni a programban. Értékadás és egyenlőségvizsgálat nem lehetséges `access` típusú paraméterekre.

A következő részben példákat láthatunk a mutatók használatára.

```
 type Dátum is record
   Nap, Hónap, Év: Integer;
 end record;
```

A hagyományos mutatók segítségével itt írhatnánk a következőt:

```
type P is access Dátum; -t,
```

de ha azt írjuk, hogy

```
type P is access all Date;
```

és legyen

```
D1, D2: P;
U: aliased Date;
```

akkor `U` egy normális változó, csak jelöljük, hogy esetleg mutatót akarunk rá állítani. Ekkor a

```
D1 := U'Access;
```

utasítást írva, a `D1` eléri az `U`-t, és `D1`-n keresztül is értéket adhatunk neki, listába fűzhetjük, stb. A következő értékadás után

```
D1.all := (2,2,1997);
```

a `D1.Nap`, illetve az `U.Nap` ugyanazt jelenti.

Megtehetjük azt is, hogy a mutató típus definícióját úgy írjuk, hogy:

```
type P is access constant Dátum;
```

ekkor read-only elérést adunk a mutatóknak, így

```
D: P;
   -- most ez konstans lesz abban az értelemben, hogy:
D := U'Access;     -- megengedett,
U.Nap := 3;        -- ez is,
D.Nap := 2;        -- ez hibás, de a következő nem:
put(D.Nap);
-- megengedett viszont az is, hogy
D := new Dátum'(2,2,1999);
```

Ha az `U`-t akarjuk konstanssá tenni, akkor azt kell írni, hogy:

```
U: aliased constant Dátum := (...);
```

Az `'Access` elérhetőségi ellenőrzést is végez, hogy a mutatott objektum élettartama legalább akkora-e, mint a mutatóé:

3.13. Példa

```
procedure Pr is
   type Dátum is record
      Nap, Hónap, Év: Integer;
   end record;
   type P is access all Dátum;
   P1: P;
   procedure B is
      Ma: aliased Dátum;
      type P1 is access all Dátum;
   ...
   P2: P := Ma'Access;              -- ez hibás
   P3: P1 := Ma'Access;             -- ez jó,
   P4: P := Ma'Unchecked_Access;    -- söt ez is,
-- de ezzel nagyon vigyázzunk, mert átvettük a felelösséget!!
   ...
```

Egy mutató típus definiálásakor előfordulhat, hogy a definíció rekurzív (például az egyszerű lista meghatározásakor). Adában viszont minden azonosítónak deklaráltnak kell lennie. Ebben az esetben a következőképpen járhatunk el:

```
type Láncszem;
-- A Láncszem azonosítót típusnévnek deklaráljuk,
-- ez lesz majd a mutató típus gyüjtötípusa.
-- Ezután definiáljuk a Láncol típust
type Láncol is access Láncszem;
type Láncszem is -- végül megadjuk a gyüjtötípus teljes
                 -- definícióját.
record
   Érték: Integer;
   Elôzô, Következô: Láncol;
end record;
```

Először a `Láncszem` azonosítót típusnévnek deklaráltuk, majd definiáltuk a `Láncol` típust. A `Láncol` típust használtuk fel a `Láncszem` gyűjtőtípus végleges meghatározásánál. A fenti típust a következőképpen használhatjuk:

```
Elsô: Láncol := null;  -- default null érték
Köv: Láncol;
Elsô := Köv;           -- másolja a mutatót
Elsô.Elôzô := Köv;
   -- a mutatott objektum komponensére egyszerü
   -- minösítéssel hivatkozhatunk
```

```
Elsô.all := Köv.all;
   -- az Elsô mutató által jelölt teljes
   -- objektumra Elsô.all formában hivatkozhatunk
C: constant Láncol := new Láncszem'(0, null, null);
```

Ugyanez lehet fordítva is, megadhatjuk először, hogy lesz egy `Láncol` típusunk, majd definiáljuk a `Láncszem` típust, s végül megadjuk a `Láncol` teljes definícióját.

```
type Láncol;
type Láncszem is
   record
   Érték: Integer;
   Elôzô, Következô: Láncol;
end record;
-- Ezután definiáljuk a Láncol típust,
type Láncol is access Láncszem;

Elsô := new Láncol'(0, null, null);
Köv := new Láncol'(0, null, Elsô); -- a Köv az Elsô-re mutat
```

Definiálhatunk konstans értékű mutatókat is. Ebben az esetben a mutató konstans és nem az objektum.

```
Konst: constant Láncol:=new Láncszem'(1000, null, null);
Konst:=null; --helytelen értékadás, mert a Konst konstans
Konst.Érték:=0; --helyes, mert az Érték mező kaphat új értéket
```

A következő típussal a bináris fa ábrázolását valósíthatjuk meg:

3.14. Példa

```
type Csúcs;
type Pcsúcs is access Csúcs;
type Csúcs is
record
   Érték: Integer;
   Bal, Jobb: Pcsúcs;
end record;

function Keres(R: in Pcsúcs; X: in Integer)
   return Boolean is
begin
   if R = null then return False;
   elsif X = R.Érték then return True;
```

```
   elsif X < R.Érték then return Keres(R.Bal, X);
   else return Keres(R.Jobb, X);
end if;
end Keres;
```

A függvény megkeres egy értéket egy bináris keresőfában.

Alprogramra is állíthatunk mutatót az Ada95-ben: ekkor a rendszer futás közben dönti el, hogy melyik függvényről van szó. Ezt *dinamikus kiválasztásnak* nevezzük.

Példa alprogramot jelölő mutató típusokra:

```
type PProcedure is access procedure (S: in Integer := 0);
P: PProcedure;
...
procedure Alprogram(I: in Integer);
...
P := Alprogram'Access;
-- Access attribútum visszaad egy mutatót az Alprogram-ra
P(13);
-- eljáráshívás 13-as paraméterrel
```

Ha `P` egy alprogram, akkor a `P'Access` az alprogram mutatója. Ha `Q` egy alprogramra mutat, akkor `Q.all` a mutatott alprogramot jelöli. Ha paraméteres alprogramra mutat, akkor `Q.all(x, y)` formában történik az alprogram meghívása, rövidítve pedig így írhatjuk: `Q(x, y)`. Az `'Access` attribútumot nem használhatjuk az előre definiált operátorok esetén (pl. `"+"'Access` esetén), csak akkor, ha a + műveletet átlapoltuk egy másik alprogrammal (ilyen esetben használhatjuk a `"+"'Access`-t).

A következő példa a függvényeket jelölő mutatók használatát mutatja be:

3.15. Példa

```
with Text_Io; use Text_io;
procedure Pfüggvény is
type Int_Függvény is access function(X: in Integer)
   return Integer;
F: array(1 .. 2) of Int_Függvény;
package Int_io is new Integer_Io(Integer); use Int_Io;

   function Négyzet(X: in Integer) return Integer is
   begin
      return X*X;
   end Négyzet;

   function Köb(X: in Integer) return Integer is
```

```
   begin
      return X*X*X;
   end Köb;

   X: Integer;
   Y: array(1 .. 2) of Integer;

-- az alprogramra állított mutatók paraméterek is lehetnek:

   procedure Demo(Fcn: in Int_Függvény) is
      X, Y: Integer;
   begin
      put_line("Demoban");
      X := 2;
      -- ...
      Y := Fcn(X);
      put(Y);
      -- ...
   end Demo;
begin
   F(1) := Négyzet'Access;
   F(2) := Köb'Access;
   X := 3;
   for I in 1 .. 2 loop
     Y(I) := F(I)(X); -- ez tulajdonképpen az F(I).all(X)
                      -- rövidítése!
     put(Y(I));
   end loop;
   -- Demo hívása pl.:
   Demo(F(1));
   -- de lehetne pl. ciklusban is
   for I in 1 .. 2 loop
      Demo(F(I));
   end loop;
   -- amikor a Demo hívja a Négyzet vagy a Köb függvényt,
   -- azt nevezzük "call back"-nek.
end Pfüggvény;
```

3.7. Feladatok

3.1. Készítsünk egy programot, mely egy adott tömbről eldönti, hogy az 1, ..., n számok permutációja-e.

3.2. Adott egy n teremből álló labirintus egy bináris mátrix formájában: a(i, j)=1 azt jelenti, hogy az i. és a j. termeket folyosó köti össze. Készítsünk egy programot, ami megadja, hogy milyen termeken keresztül juthatunk el egy adott teremből egy adott másik terembe.

3.3. Az életjátékban a sejtek egy négyzetrácsba rendezve helyezkednek el. A sejteknek két állapotuk lehet: élő vagy holt. A sejtállományt egy mátrixszal ábrázoljuk. A sejteknek, kivéve a szélén álló sejteknek, nyolc szomszédjuk van. Egy holt sejt helyén akkor keletkezik új sejt, ha pontosan három élő szomszédja van, illetve egy élő sejt csak akkor marad életben, ha két vagy három szomszédja van, különben kihal. Készítsünk programot, ami egy tetszőleges kezdeti állapotból kiindulva kiírja egy sejtállomány megadott számú állapotváltozását.

3.4. Írjunk egy sablont kétszeresen láncolt lista megvalósítására, majd ennek segítségével hozzunk létre egy olyan listát, amelyben az elemeket növekvő sorrendben helyezzük el.

3.5. Írjunk egy bináris fa típust létrehozó csomagot. A csomag segítségével:

- írjuk ki hány elem van egy fában;
- a bejáró algoritmusok segítségével írjuk ki egy fa elemeit;
- hozzunk létre keresőfát és rendezzük egy tömb elemeit a fa segítségével.

4. Fejezet

Utasítások

Az Ada programegységek törzse egy deklarációs részt és egy *utasítás-sorozat*ot tartalmaz. Ezek az utasítások a leírásuk sorrendjében hajtódnak végre, s ezt a sorrendet csak egy vezérlési szerkezet vagy egy kivétel fellépése változtathatja meg. Ebben a fejezetben csak az Ada95 nyelv alapvető utasításait ismertetjük, pl. a kivételkezeléssel, párhuzamos programok készítésével kapcsolatos nyelvi elemeket a könyv megfelelő fejezeteiben mutatjuk be.

4.1. Egyszerű utasítások

4.1.1. Az értékadás

Az értékadás arra szolgál, hogy egy kifejezés értékét egy változóhoz rendeljük.

```
i:= 5;
```

Az értékadásban az érték és a változó típusának meg kell egyeznie[1]. Ha a kifejezés értéke nem felel meg a változó altípusának, akkor fordítási vagy futási időben hibaüzenetet (pl. `Constraint_Error`) kapunk. Az Ada fordító szigorúan ellenőrzi az értékadásban szereplő változók és kifejezések típusainak egyezését és lehetőség szerint már fordítási időben hibaüzenettel jelzi az eltérést. Az értékadás a korlátozott típusú változóktól eltekintve minden változóra alkalmazható.

4.1. Példa

```
I,J : Integer range 1 .. 10;
K   : Integer range 1 .. 20;
I := J;  --  azonos tartományúak
K := J;  --  összeegyeztethetö tartományúak
```

[1]Helyes kifejezések felépítésének szabályairól, valamint objektumok inicializálásáról, illesztéséről és lezárásáról részletes ismertetést talál az érdeklődő olvasó a könyv megfelelő fejezeteiben.

```
J := K;  -- ha  K > 10, akkor futási időben
         -- Constraint_Error hibaüzenetet vált ki
```

Az értékadás tömbökre és résztömbökre is alkalmazható, ilyenkor a két tömb elemtípusának és az elemszámnak meg kell egyeznie, az indexhatárok eltérhetnek.

4.2. Példa

```
A : String(1 .. 31);
B : String(3 .. 33);

A:= B;  -- azonos elemszámúak
A(1 .. 15):= "ma szép idő van";
A(3 .. 17):= A(1 .. 15); -- A(1 .. 17) = "mama szép idő van"
```

4.1.2. A `null` utasítás

Az Ada95 nyelvben üres utasítássorozatot nem adhatunk meg, amennyiben a program változóit nem kívánjuk megváltoztatni – állapotváltozást nem akarunk előidézni – akkor az egyelemű, a `null` utasítást tartalmazó utasítássorozatot kell megadnunk.

4.1.3. Alprogramok hívása és a `return` utasítás

Alprogramokat nevük és aktuális paramétereik megadásával hívhatunk[2]. Ha a paraméterek listája üres, akkor a paraméterlistát közrefogó zárójelpárt elhagyjuk. Az aktuális paramétereket vesszővel választjuk el egymástól, megadhatjuk őket a formális paraméterek sorrendje vagy elnevezése szerint. Egyes paraméterek elhagyhatóak.

```
Ada.Text_IO.Put("Kiírandó szöveg");
Ada.Text_IO.Get(ch); -- ahol ch egy karakter típusú változó
Ada.Text_IO.New_Line;
```

A `return` utasítás hatására az alprogram végrehajtása befejeződik. Ha az alprogram függvény, akkor a `return` utasításban adjuk meg a függvény eredményét is.

```
return (x+y);
```

A függvény eredményének típusára az Ada nem tesz megkötést, így az nemcsak skalár, hanem összetett típus is lehet.

[2] Az alprogramokat részletesen tárgyaljuk a könyv 5. fejezetében.

4.2. Összetett utasítások

Az összetett utasítások a programkonstrukciók kódolására használhatóak. A programkonstrukciók egymásba ágyazhatóak, így egy adott programnyelven történő kódolásuk esetén mindig törekedni kell arra, hogy világosan meghatározható legyen a szerkezet eleje és vége. Az Ada nyelvben a programszerkezetet bevezető kulcsszó megjelenik a szerkezetet lezáró utasításban, nagyban növelve a nyelv szintaktikus biztonságát. Ennek megfelelően nincs szükség a programszerkezetek esetén `begin`,`end` utasításpár alkalmazására, mint pl. Pascalban, Modula-2-ben, vagy C-ben.

4.2.1. Elágazás: `if-then-elsif-else`, `case`

Elágazások esetén a megadott utasítássorozatok közül legfeljebb az egyik végrehajtására kerül sor. Elágazások belsejébe tetszőleges mélységben további összetett utasítások, pl. újabb elágazások is beágyazhatóak.

Az `if` utasítás

Elágazások kódolásához az `if` utasítást használhatjuk. Az `if` kulcsszó után álló logikai kifejezés értéke szerint az egyik vagy a másik ágban szereplő utasítássorozat hajtódik végre. Ha a kifejezés értéke igaz, akkor a `then` kulcsszót közvetlenül követő utasítássorozat hajtódik végre, különben az `else` ág kerül végrehajtásra. Ha az `else` ág egyetlen `null` utasításból állna, akkor ezt az ágat elhagyhatjuk az `else` kulcsszóval együtt. Ha az `else` ágban újabb elágazást szeretnénk kódolni, akkor az `else` és az őt követő `if` kulcsszavakat összevonhatjuk az `elsif` kulcsszó alkalmazásával. Ebben az esetben a kód könnyebben olvasható, módosítható. Megjegyezzük, hogy egy többszörösen egymásba ágyazott `if` szerkezet esetén à programszöveg esetleges módosítása során tekintettel kell lenni arra, hogy a külső elágazás feltételének módosítása kihat a beágyazott elágazás szemantikájára is.

4.3. Példa

```
if Hónap = December and Nap = 31 then
  Hónap := Január; Nap := 1; Év := Év + 1;
end if;
if Beljebb_Kezd then
    Bal_Margó_Vizsgálat;
    Eltolás_Balra;
elsif Kijjebb_Kezd  then
    Eltolás_Jobbra;
else
    Új_Sor;
end if;
```

A `case` utasítás

Esetszétválasztással is kódolhatunk elágazást, ha az esetek szétválasztására egy diszkrét típusú kifejezés értékét használjuk. A `case` utasításban vagy használni kell az `others` ágat, vagy pedig a kifejezés minden lehetséges értékét fel kell sorolni a `when` ágak között, és mindegyiket pontosan egyszer. Egy-egy ághoz több érték is tartozhat, értékek tartományát, a diszkrét típus egy altípusát is megadhatjuk. Az egy-egy ághoz tartozó értékek halmazát meghatározó kifejezéseknek statikusnak kell lenniük, értékük fordítási időben meghatározásra kerül[3].Ha az `others` szót használjuk, akkor annak az utolsó ágban kell lennie.

4.4. Példa

```
subtype Húszig is INTEGER range 1 .. 20;
Érték : Húszig;
--
case Érték is
  when 1 .. 3 | 5 | 7 | 11 | 13 | 17 | 19 => Felbonthatatlan;
  when others => null;
end case;

case Egyes_Bitek_Száma(Számláló) is
  when 1      => Módosít(1);
  when 2      => Módosít(2);
  when 3 | 4   => Töröl(1); Töröl(2);
  when others => raise Hiba;
end case;

case Nap is
   when Hétfő => Kezdeti_Egyenleg;
   when Péntek => Záró_Egyenleg;
   when Kedd .. Csütörtök => Jelentés(Nap);
   when Szombat | Vasárnap => null;
end case;
```

4.2.2. Ciklus utasítások: `for`, `while`, `exit`

A ciklusnak három formája használatos. Az egyik a tradicionális iteráció ciklusváltozó segítségével. A ciklusváltozó a ciklus lokális konstansa, amit a ciklus első sorában deklarálunk. A változó rendre felveszi az `in` kulcsszó után megadott diszkrét altípushoz tartozó értékeket rendezés szerint növekvő, ill.

[3]`case` utasítás esetén hatékonysági megfontolások következtében a kifejezés értéke nem feltétlenül a felsorolás sorrendjében kerül összehasonlításra az egyes ágak előtt szereplő értékhalmazokkal.

a `reverse` kulcsszó megadása esetén csökkenő sorrendben. A ciklusváltozó értéke a ciklus magjában nem változtatható meg. A `for` ciklus üres tartomány esetén egyszer sem hajtja végre a ciklusmagot.

4.5. Példa

```
for I in 1 ..  Elemek_Száma loop
   Beolvas(Új_Elem);
   Összefésül(Új_Elem.File);
exit when Új_Elem = Záró_Elem;
end loop;

for J in reverse Tároló'Range loop
  if Tároló(J) /= SPACE then
     PUT(Tároló(J));
  end if;
end loop;
```

A ciklus második formája a while típusú ciklus. Ebben az esetben a ciklusmagot újra meg újra végrehajtjuk, amíg a ciklusfeltétel igaz. Ha a feltétel kezdetben hamis, akkor a ciklusmag egyszer sem kerül végrehajtásra.

4.6. Példa

```
Összeg := 0;
while  Következô  /= null loop;
 Összeg := Összeg +  Következô.Érték;
   Következô := Következô.Rákövetkezô;
end loop;

while Ajánlat(N).Ár < Határérték loop
  Értékeld(Ajánlat(N));
  N := N + 1;
 end loop;
```

A harmadik forma egy feltétel nélküli ciklus, aminek a megszakítását az `exit` utasítással érhetjük el (ld. később).

4.7. Példa

```
loop
   --
exit when Feltétel;
   --
end loop;
```

A ciklusnak nevet is adhatunk. Ennek formája az, hogy a ciklusfej előtt, attól kettősponttal elválasztva leírunk egy azonosítót. A ciklusnévnek pl.

az `exit` utasításban, és egymásbaágyazott ciklusok esetén a ciklusváltozó azonosításakor van szerepe.

```
A1 : for I in ... loop
A2 : for I in ... loop
     ... B(A1.I) ...
     end loop A2;
     end loop A1;
```

4.2.3. A blokk és a `declare` utasítás

Az Ada a blokkstruktúrájú nyelvek leszármazottja. Blokkokban lokális deklarációkat helyezhetünk el[4]. Blokkot képezni az Ada programban nemcsak alprogram kialakításával, hanem `declare` utasítással is lehet.

A `declare` utasítás a következő helyzetekben használható előnyösen:

- Ha egy nagy adatstruktúrára van szükség a program egy rövid szakaszán. Ebben az esetben a struktúra deklarálható egy blokkban, így a memóriát a program csak arra az időre terheli meg, amíg a struktúra aktív.

 4.8. Példa
  ```
  declare
   Temp : T;
  begin
   Temp := A; A := B; B := Temp;
  end;
  ```

- Az Ada megengedi olyan objektumok definiálását, amelyek méretét futási időben számítjuk ki. Van úgy, hogy a méretet nem lehet egy kifejezéssel megadni, egy kevés számítást is el kell végezni a változó deklarációja előtt. Egy lehetséges megoldás ilyenkor a `declare` utasítás használata.

 4.9. Példa
  ```
  A: Vektor(N);
  ...
  M := A'First;
  while M /= A'Last  and then A(M) /= 0 loop
      M := M + 1;
  end loop;
  declare
  ```

[4] A lokális deklaráció fogalmát a láthatóságról szóló fejezetben tárgyaljuk.

```
    B:Vektor(M);
  begin
     ...
  end;
```

- Már egy deklarációs rész nélküli blokktörzs is lehetővé teszi, hogy a blokkban kivételkezelő programrészt helyezzünk el.

4.10. Példa

```
begin
   A := C / B;
exception
  when Constraint_Error => A := Integer'Last;
end;
```

A blokkban használhatjuk annak lokális azonosítóit, és a blokkot magába foglaló egységben látható azonosítókat, mint globális neveket. Ha névazonosság fordul elő, szelekciós jelöléssel (pl. `X.A`) hivatkozhatunk a kívánt objektumra.

4.11. Példa

```
Külsô : declare
 I,J:Integer;
begin
  -- I,J:Integer  közvetlenül elérhetö
  Belsô  : declare
  I: Saját_Int;
  begin
    -- I:Saját_Int és J:INTEGER közvetlenül elérhetö, de
    -- I:Integer  csak Külsô.I alakban érhetö el
  end  Belsô;
    --
end Külsô;
```

Az `exit` utasítás

Az `exit` utasítás az egyszerű utasítások közé tartozik, ciklusmagokban helyezhető el. Hatására a vezérlés a ciklus utáni első utasításra adódik át.

4.12. Példa

```
loop
 Ada.Text_IO.Get(Ch);
   exit when Ch = '.';
 if Ch = '(' or CH = ')' then
```

```
   Text_IO.Put(Ch);}
 end if;
end loop;

K: loop
      --
    loop
       --
      exit K when Feltétel;
         --
    end loop;
         --
   end loop K;
```

4.3. A goto utasítás

A goto utasításra a nyelvben valójában nincsen nagy szükség. Azért vették be az utasításkészletbe, mert van olyan helyzet, amikor a kódot rövidebbé lehet tenni goto használatával.

4.13. Példa

```
<<COMPARE>>
 if A(I) < Elem then
   if Balszomszéd(I) /= 0 then
      I := Balszomszéd(I);
      goto COMPARE;
   end if;
   --  utasítások
 end if;
```

4.4. Feladatok

4.1. Módosítsa a 4.3. példa programrészletét oly módon, hogy az az eredetivel ekvivalens maradjon, de elsif utasítás ne szerepeljen benne!

4.2. Írjon a 4.4. példában bemutatott elágazásokkal azonos hatású programrészleteket case helyett az if utasítás használatával.

4.3. Írja át az alábbi for ciklust while ciklusra!

```
 for i in 2 .. vlen loop
   if v ( i ) > max then
      max := v(i);
```

```
    mpos := i;
  end if;
end loop;
```

4.4. Írja át 4.5. példában szereplő ciklust `while` ciklusra!

4.5. Készítsen a `goto` utasítás használata nélkül a 4.13. példában megadott programrészlettel ekvivalens programot!

5. Fejezet

Alprogramok

Az alprogramok olyan programegységek, amelyek végrehajtását meghívásukkal lehet kezdeményezni. A programozási nyelvekben arra használjuk őket, hogy különböző számításokat elkülönítsünk egymástól: egy alprogram egy jól körülhatárolható számítási feladatot old meg. Egy ilyen számítás lehet például egy nagyobb számítási feladat egy részfeladatának megoldása, vagy egy típus egy műveletének megvalósítása, de alprogram lehet egy algoritmust vagy egy matematikai függvényt kódoló, gyakran felhasznált könyvtári egység is. Bármi is a feladata, az alprogram a programkód feldarabolásának egy fontos eszköze: ugyanúgy, ahogy egy bonyolult feladatot felbonthatunk egyszerűbb részfeladatokra, egy komplex számítást leíró programot felépíthetünk egymást meghívó alprogramokból.

Az Adában – sok más programozási nyelvhez hasonlóan – kétféle *alprogramot* különböztetünk meg: *eljárásokat* és *függvényeket*. Az eljárások általában a változók által kifeszített téren (állapottéren) vagy a program környezetén elvégzendő transzformációkat adnak meg. A függvények valamilyen értéket állítanak elő, de transzformációt nem végeznek, sem a program változóinak értékére, sem a program környezetére nincsenek semmilyen hatással: úgy szokták mondani, hogy a függvényeknek nincs *mellékhatásuk*. Ez sajnos leginkább programozói stílus kérdése: a procedurális programozási nyelvek, mint például az Ada, nem kötelezik a programozót ennek a fontos programozáselméleti szabálynak a betartására, ezzel ugyanis a nyelv rugalmasságát csökkentenék. Egy jó programozó azonban mindig törekszik a függvényeket arra használni, amire valók: egy bonyolult vagy rekurzív kifejezés, egy matematikai formula, vagy egy lekérdező művelet megvalósítására.[1]

[1]Egyes programozási nyelvekben, például C-ben, C++-ban, Java-ban, az eljárás és a függvény fogalma nem válik szét: programozáselméleti szempontból ez helytelen, noha tény, hogy ezzel a nyelv egyszerűbbé, könnyebben megtanulhatóvá válik. A védett egységekkel foglalkozó fejezetben (16.) látni fogjuk majd, hogy az Ada feltételezi a programozóról a függvények fegyelmezett használatát, így kihasználva a függvények mellékhatásmentességét, hatékony, mégis biztonságos eszközöket ad a programozónak cserébe – a párhuzamos programok írása terén.

5.1. Példa

```
function Maximum_Hely( V: Vektor ) return Index is
  M: Index := V'First;
begin
  for I in V'Range loop
    if V(M) < V(I) then M := I; end if;
  end loop;
  return M;
end Maximum_Hely;

procedure Rendez( V: in out Vektor ) is
  E: Elem;
  I: Index;
begin
  for J in reverse V'range loop
    I := Maximum_Hely(V(V'First..J));
    E := V(I);
    V(I) := V(J);
    V(J) := E;
  end loop;
end Rendez;
```

Az Adában az eljárásokat a `procedure`, a függvényeket a `function` kulcsszó vezeti be.

Egy eljárás végrehajtása olyan, mint egy utasítás, egy függvény végrehajtása pedig egy kifejezés értékének meghatározása. Az eljárás hívása egy az utasítássorozatot alkotó utasítások között, míg a függvény hívása egy kifejezés, például egy részkifejezés egy összetett kifejezésben:

```
Egyenest_Rajzol(Kezdôpont,Végpont);   -- eljárás hívása
if Elemek_Száma(Halmaz) > 0 then ...  -- függvény hívása
```

Mint azt a fenti példákból látjuk, az alprogramoknak *paramétereken* keresztül adhatjuk át a működésükhöz szükséges információt, és ők is ezeken a paramétereken (illetve függvények esetén az ún. *visszatérési értéken*) keresztül juttatják el a számításuk eredményét a hívó programegységnek. Azt, hogy az információ hogyan és merre áramlik a paraméterek átadása során, a paraméterek módja és a paraméterátadás technikája határozza meg. (Lásd 5.1.) A lényeg az, hogy az alprogram megadásakor definiált, úgynevezett *formális paramétereknek* az alprogram hívásakor – a hívás helyén értelmes – változókat vagy kifejezéseket, úgynevezett *aktuális paramétereket* feleltetünk meg. Az alprogram ezekkel az aktuális paraméterekkel fogja elvégezni a benne megfogalmazott számítást.

A függvények visszatérési értékének típusát a `return` kulcsszó után kell megadni. A függvénynek tartalmaznia kell (egy vagy több) `return` utasítást

is. Ez az utasítás használható a függvény visszatérési értékének megadására. Az utasítás hatására a függvény végrehajtása be is fejeződik.[2] Paraméter nélküli `return` utasítás eljárásokban is állhat: hatására az eljárás végetér.

Egy alprogram megadásánál ezek szerint definiálni kell, hogy az alprogram hogyan aktivizálható, mi a kapcsolata a környezetével, azaz mi az interfésze a külvilág felé. Ezen kívül meg kell adni az alprogram által végrehajtandó utasítássorozatot (ezzel definiáljuk az alprogram által elvégzendő számítást), valamint az alprogramban használt kisegítő változókat. Ezek a kisegítő változók csak az alprogram végrehajtásának ideje alatt élnek, az alprogramon kívül nem is használhatók. Ők az alprogram *lokális változói.* (Megjegyzés: Előfordulhat – rekurzív vagy párhuzamos programok esetén – , hogy egy időpillanatban egy alprogram több példányban is végrehajtódik: ahogy ezt mondani szokás, az alprogramok reentránsak. Ilyenkor minden egyes végrehajtási példány saját lokális változó példányokkal rendelkezik.)

Van, hogy az alprogram számára szükséges információt (vagy annak egy részét) nem paramétereken keresztül adjuk át, hanem úgynevezett *globális változók* segítségével. A globális változók olyan változók, amelyek – bár nem az alprogramban voltak definiálva, de a változó-deklarációkra vonatkozó láthatósági, hatásköri szabályok értelmében, lásd 8. – érvényesek, használhatók az alprogramon belül. Ezek az alprogram hívásának helyén megfelelő értékkel rendelkezhetnek, így juttatva el a kívánt információt az alprogram utasításai számára. A globális változók használata megkönnyítheti az alprogramok hívását, hiszen kevesebb paramétert kell átadnunk, viszont ne feledjük, hogy használatuk nehezen olvashatóvá, nehezen érthetővé teszi a kódot: ha nincs valami különleges okunk az ellenkezőjére, kerüljük használatukat.

Az Adában a paraméter nélküli eljárások különös jelentőséggel bírnak: minden program *főprogramja* egy paraméter nélküli eljárás kell legyen. Hasonlóan, minden paraméter nélküli eljárás lehet egy főprogram, ha nincs más programegységbe beágyazva, és az általa használt, de rajta kívül definiált programobjektumokat (globális változókat, alprogramokat, csomagokat, stb.) elérhetővé tesszük számára (lásd 8.):

5.2. Példa

```
with Ada.Integer_Text_IO, Text_IO;
procedure Rendezés is
  subtype Elem is Integer;
  subtype Index is Positive;
  type Vektor is array (Index range <>) of Elem;

  function Maximum_Hely( V: Vektor ) return Index is
    ... -- mint fent
```

[2] Ha egy függvény végrehajtása úgy ér véget, hogy nem került sor `return` utasításra, futási idejű hiba (`Program_Error` kivétel, lásd 126. o.) váltódik ki.

```
  procedure Rendez( V: in out Vektor ) is
    ... -- mint fent

  V: Vektor(1..10) := (3,7,2,5,8,2,4,9,0,1);
begin
  Rendez(V);
  for I in V'Range loop
    Ada.Integer_Text_IO.Put(V(I));
  end loop;
  Text_IO.New_Line;
end Rendezés;
```

Ebben a fejezetben bemutatjuk az Ada azon tulajdonságát is, hogy különböző alprogramok hogyan lehetnek azonos nevűek, azaz hogyan lehet az alprogramneveket *átlapolni.* Az erre vonatkozó „ökölszabály" az, hogy az alprogram hívásakor egyértelműnek kell lenni (például a paraméterek száma és típusa alapján), hogy melyik alprogramra gondoltunk az ugyanolyan nevűek közül. További érdekesség, hogy még a predefinit operátorokat, például a `"+"` műveletet is át lehet lapolni.

5.1. Paraméterek

Amikor egy alprogramot specifikálunk, megadjuk többek között azt is, hogy mi a paraméterlistája, azaz hány formális paramétere van, és mi azoknak a típusa. (A paraméterlista ennél több információt is tartalmaz, mint majd ezt látni fogjuk.)

```
procedure Szia_Világ
function Fordítva ( S: String ) return String
function Maximum ( X: Integer; Y: Integer ) return Integer
function Maximum ( X, Y: Integer ) return Integer
```

Az első alprogram egy paraméter nélküli eljárás. Ha egy alprogramnak nincs paramétere, a paraméterlistát határoló zárójelpár is elmarad.

Az azonos típusú formális paraméterek definícióját össze lehet vonni.[3] A `Maximum` függvény két specifikációja teljesen kompatíbilis.[4]

Mint láthatjuk a `Fordítva` függvény példáján, egy formális paraméterhez (vagy függvény visszatérési értékhez) elég egy megszorítatlan (például meghatározatlan diszkriminánsú) típust írni. Ilyenkor a paraméter pontos altípusát az aktuális paraméter fogja meghatározni. Ez a szabály lehetővé teszi,

[3] Egész pontosan: az azonos típusú és módú paraméterekét. A paraméterek módjáról lásd: 5.1.1.

[4] Az Ada83-ban szigorúbbak voltak a szabályok. Azok szerint ez a két specifikáció nem teljesen kompatíbilis.

hogy ezt a függvényt, vagy például a `Text_IO.Put_Line` eljárást különböző hosszúságú String-ekkel hívhassuk meg.[5]

```
procedure Put_Line(Item : in  String);  -- az Ada.Text_IO-ból

Put_Line(Fordítva("indulagörögaludni"));
Put_Line(Fordítva("seygü")&" vagy!");
```

Egy függvény visszatérési értéke bármilyen típusú lehet, akár tömb vagy rekord is. Az összetett típusú értéket visszaadó függvények hívására alkalmazhatóak a típus típusműveletei is. Tehát, ha például `F` egy tömb értékű függvény, akkor szabályos az `F(X)(1)` indexelés, ami az eredménytömb objektum 1-es indexű elemét adja meg. Ugyanígy, ha `F` egy rekord értékű függvény, akkor szabályos az `F(X).S` alakú szelekciós kifejezés (ha `S` a rekord egy mezője).

Az aktuális és formális paraméterek megfeleltetésére ugyanazok a szabályok érvényesek, mint az értékadásra: az aktuális és a formális objektum típusának meg kell egyezni. Paraméterátadáskor ellenőrzésre kerül az is, hogy az átadott értékek megfelelnek-e az altípus-megszorításoknak.

5.1.1. A paraméterek módja

A paraméterek háromfélék lehetnek. Az a paraméter, amit `in` módúnak definiálunk, az alprogramba juttat információt a hívóból. Az `out` módú paraméterek ezzel szemben azért vannak, hogy az alprogramban kiszámolt értéket a hívónak át tudjuk adni. Végül vannak az `in out` módú paraméterek: ezek az alprogram meghívásakor információt adnak az alprogramnak a hívótól, majd az alprogram befejeződésekor információt adnak a hívónak az alprogramtól.

```
procedure Közös (A, B: in Positive; Lnko, Lkkt: out Positive)
procedure Rendez (V: in out Vektor)
```

Az `A` és a `B` bemenő paramétere a `Közös` eljárásnak, míg az `Lnko` és az `Lkkt` kimenő paraméterek. A `Rendez` eljárás `V` paramétere mind bemenő, mind kimenő: az eredményt a bemenetként kapott vektorban adjuk vissza.

Amikor egy alprogramot meghívunk, a formális paraméterek módja meghatározza, hogy mit adhatunk át aktuális paraméternek. Egy `in` paraméternek átadhatunk egy tetszőleges kifejezést (például egy változót is): a kifejezés értéke lesz az aktuális paraméter. Egy `out` vagy `in out` paraméterként viszont csak egy változót adhatunk át: ebbe a változóba kerül majd az alprogram által kiszámított érték (illetve `in out` mód esetén ez a változó tartalmazza a bemenő paraméterértéket is).

[5] A polimorfizmus (többalakúság) szót – az objektum-orientált terminológiánál (lásd 12.1.) – tágabban értelmezve úgy fogalmazhatunk, hogy ezáltal polimorf alprogramokat készíthetünk.

```
Közös(a,a+42,x,y);             -- helyes
Közös(Maximum_Hely(V),42,x,y); -- helyes
Közös(30,12,20,10);            -- helytelen
Rendez(Vous);                  -- helyes
Rendez((2,6,4,8,2,9));         -- helytelen
```

A formális paraméterek módja kihat arra, hogy az alprogramon belül hogyan használhatjuk őket. Egy `in` típusú formális paraméternek nem adhatunk értéket, csak olvashatjuk. Egy `out` módú paramétert lehet írni, és ha már kapott valamilyen értéket az alprogramon belül, lehet olvasni is.[6] Egy `in out` módú paraméter értékét olvashatjuk és írhatjuk is.

5.3. Példa

```
procedure E ( Vi: in Integer; Vo: out Integer;
              Vio: in out Integer ) is
begin
  Vio := Vi + Vo;   -- helytelen, Vo-t nem olvashatjuk
  Vi := Vio;        -- helytelen, Vi-t nem írhatjuk
  Vo := Vi;         -- helyes
  Vo := 2*Vo+Vio;   -- helyes, Vo már kapott értéket
end E;
```

Törekedjünk arra, hogy az általunk definiált formális paraméterek módja valóban az információáramlás irányát írja le. Ne az legyen a szempont abban, hogy egy paramétert `in out` módúnak választunk, hogy kényelmes lenne az alprogram törzsében írni az értékét, mert így nehezen olvasható, nehezen érthető, sőt, könnyen hibás programot is kaphatunk.

```
procedure Közös ( A, B: in out Positive;
                  Lnko, Lkkt: out Positive ) is
begin
  Lkkt := A * B;
  while A /= B loop
    if A > B then A := A - B; else B := B - A; end if;
  end loop;
  Lnko := A;
  Lkkt := Lkkt / Lnko;
end Közös;
```

Ennek az eljárásnak a specifikációja olyan csúnya, hogy azt nem feledtetheti az sem, hogy megspóroltunk két változódeklarációt. A programozói szempontból szép megoldás az `A` és `B` változókat `in` módúnak definiálja, és két lokális változót (`A` és `B` értékével inicializálva) használ a ciklushoz helyettük.

[6] Az Ada83-ban szigorúbb szabály volt erre nézve. Szimmetrikusan az `in` módú paraméterekkel, az `out` módú paramétereket csak írni lehetett, olvasni nem.

5.4. Példa

```
procedure Közös ( A, B: in Positive;
                  Lnko, Lkkt: out Positive ) is
  X: Positive := A;
  Y: Positive := B;
begin
  while X /= Y loop
    if X > Y then X := X - Y; else Y := Y - X; end if;
  end loop;
  Lnko := X;
  Lkkt := A * B / Lnko;
end Közös;
```

(Megjegyzés: Ha az eljárást `Közös(I,J,X,Y);` alakban hívja valaki, az első esetben könnyen elkerülheti a figyelmét, hogy az eljárás `I` és `J` értékét megváltoztatja, hiszen ez nem „természetes". És már elő is állított az illető egy „bug"-ot, egy hibát, amit esetleg csak hosszadalmas nyomkövetéssel talál meg, hiszen amit leírt, az jó, csak nem azt jelenti, amit ő gondolt.)

Egy formális paraméter definiálásakor nem feltétlenül kell kiírni a módját: amennyiben semmilyen módot nem írunk, az alapértelmezett `in` mód rendelődik a paraméterhez.

```
procedure Egyenest_Rajzol ( Kezdôpont, Végpont: Pont )
```

Ha két alprogram specifikáció csak abban tér el, hogy bizonyos formális paraméterek az egyikben specifikáltan `in` módúak, míg a másikban az alapértelmezett mód miatt, a két specifikáció teljesen kompatíbilis.[7]

Függvények formális paramétere csak `in` módú lehet. Ez a szabály is azt sugallja, hogy a függvényeknek ne legyen mellékhatásuk.

Egy alprogram aktuális paramétere lehet egy típuskonverzió is. Ekkor az `in` és `in out` módú paraméterek esetén a paraméterátadás előtt, `in out` és `out` paraméterek esetén az értékek visszaadásakor történik meg a konvertálás.

5.1.2. Másolat és hivatkozás szerinti paraméterátadás

Az Adában a paraméterek átadására két technika van: a másolat szerinti és a hivatkozás szerinti. A *másolat szerinti paraméterátadás* azt jelenti, hogy az aktuális és formális paraméterek különböző objektumot jelölnek, és a kettő közötti információáramlás az alprogram végrehajtása előtt, illetve

[7] Az Ada83 szabályai szigorúbbak, azok szerint két ilyen specifikáció már nem teljesen kompatíbilis.

után, egy értékmásolással történik. Ezzel szemben a *hivatkozás szerinti paraméterátadásnál* az aktuális és a formális paraméter ugyanazt az objektumot jelöli (a formális paraméter az aktuális paraméter által jelölt objektum egy nézete lesz). A hivatkozás szerinti paraméterátadás esetén az alprogram végrehajtása során a formális paraméterre vonatkozó hivatkozások (legyen az olvasás a formális paraméterből, vagy a formális paraméter írása) egyenesen az aktuális paraméterre vonatkoznak. Az, hogy egy paraméternél melyik átadási technika használandó, a paraméter típusának a függvénye.

Hasonló szabályok vonatkoznak a függvények visszatérési értékére: megkülönböztetjük a *másolat-visszaadást* és a *hivatkozás-visszaadást*. Egy függvénynél a visszaadás technikáját – ugyanúgy, mint a paraméterátadás esetén – a visszaadott típus határozza meg.

Egy alprogram hívása esetén a következő módon történik az aktuális és formális paraméterek közti információáramlás. Ha egy paramétert hivatkozás szerint kell átadni, (mind a három paramétermód esetén) létrejön az aktuális paraméter által jelölt objektumhoz egy olyan nézet, ami a formális paraméter típusának megfelel. Ez a nézet rendelődik hozzá a formális paraméterhez. Mint látni fogjuk, a jelölt típusok (lásd 12.) hivatkozás szerint adódnak át paraméternek:

5.5. Példa

```
with Ada.Integer_Text_IO;
procedure Hivatkozás_Szerint is
  type T is tagged record      -- T jelölt típus
    D: Integer := 42;
  end record;
  X: T;
  procedure P ( Y: in out T ) is
  begin
    Y.d := Y.d + 1;        -- az Y és az X ugyanazt
    X.d := X.d + 1;        -- az objektumot jelentik
  end P;
begin
  P(X);
  Ada.Integer_Text_IO.Put(X.D);  -- 44 az eredmény
end Hivatkozás_Szerint;
```

Más a helyzet a másolat szerint átadott paraméterekkel. Egy `in` vagy `in out` módú formális paraméterhez létrejön egy megfelelő típusú objektum, és ebbe az objektumba bemásolódik az aktuális paraméter értéke. (Az aktuális paraméter altípusa implicit módon konvertálódik a formális paraméter altípusára.) Az `out` módú paraméterekhez szintén létrejön egy objektum, de ez az objektum nem inicializálódik az aktuális paraméter

értékével.[8] Az alprogram befejeztével az `in out` és `out` módú formális paraméterek értéke visszaíródik az aktuális paraméterbe, megint a megfelelő altípuskonverzió után.

5.6. Példa

```
with Ada.Integer_Text_IO;
procedure Másolat_Szerint is
  X: Integer := 42;
  procedure P ( Y: in out Integer ) is
  begin
    Y := Y + 1;  -- az Y olyan, mint egy lokális változó
    X := X + 1;  -- az X és az Y különbözô objektumok
  end P;
begin
  P(X);
  Ada.Integer_Text_IO.Put(X);  -- 43 az eredmény
end Másolat_Szerint;
```

Típusok csoportosítása paraméterátadási technika szerint

Azt, hogy egy paraméter melyik technika szerint adódik át, a paraméter típusa határozza meg. A nyelv rögzíti, hogy mely típusú paramétereket kell az egyik és melyeket a másik paraméterátadási technika szerint átadni. Az olyan típusú paraméterek átadási technikája, amelyek nem szerepelnek egyik felsorolásban sem, nincs meghatározva, implementációfüggő – az adott fordítóprogram döntheti el, melyik mechanizmust választja.

Másolat szerint: az alaptípusok (skalár és mutató típusok) tartoznak ebbe a kategóriába, valamint az olyan átlátszatlan típusok (és leszármazottjaik), amelyeknek a teljes típusa (azaz a reprezentációja) érték szerint átadandó típus.

Hivatkozás szerint: a jelölt típusok, a taszk és védett típusok, azok a nem átlátszatlan típusok, amelyek deklarációjában szerepel a `limited` kulcsszó, azok az összetett típusok, amelyek valamelyik komponense, és az olyan átlátszatlan típusok, amelyek teljes típusa hivatkozás szerint átadódó típusú, valamint mindezen típusok leszármazottjai sorolandók ide.

Függvények visszatérési értéke

A paraméterátadáshoz hasonlóan a függvények visszatérési értéke is kétféleképp juthat el a függvényhívást kezdeményező kifejezésbe: másolat szerinti

[8] Bizonyos esetekben egy out módú paraméter is inicializálódik, pontosan úgy, mint egy in out módú: ha a paraméter mutató típusú, vagy olyan összetett típusú, amelynek vagy diszkriminánsa, vagy alapértelmezett értékkel rendelkező komponense van.

és hivatkozás szerinti mechanizmussal. Ebben az esetben is a visszatérési érték típusa az, ami alapján a választás megtörténik. A típusokra ugyanazok a szabályok érvényesek, mint a paraméterek esetén, az egyetlen különbséget a jelölt típusok jelentik. A jelölt típusok mindig hivatkozás szerint adódnak át, ha paraméterek, de visszatérési értékként csak akkor hivatkozás szerintiek, amikor korlátozottak (tartalmazzák a `limited` kulcsszót a deklarációjukban).

Paraméterátadási technikák más nyelvekben

A programozási nyelvekben leggyakrabban az érték szerinti, a cím szerinti, a hivatkozás szerinti, a név szerinti és az eredmény szerinti paraméterátadási technikák használatosak. Vizsgáljuk meg, hogy ezek a fogalmak hogyan kapcsolódnak az Ada paraméterátadási mechanizmusához.

Egy `in` módú paraméter másolat szerinti átadása az érték szerinti paraméterátadásnak felel meg: létrejön a formális paraméterhez egy lokális változó, ami az aktuális paraméter értékével inicializálódik. Sok nyelvben (például C-ben, C++-ban, Modula-2-ben, Java-ban) azonban ezt a létrejött változót teljes jogú lokális változóként használhatjuk, míg Adában csak arra, hogy a benne tárolt értéket kiolvassuk.[9]

A cím szerinti paraméterátadás az Adában hivatkozás szerintinek nevezettre hasonlít, de a paraméterek módja és az ezekből adódó megszorítások megkülönböztetik őket. Az eredmény szerinti paraméterátadás az `in out` (és részben az `out`) módú paraméterek másolat szerinti átadásával egyenértékű. Végezetül megállapíthatjuk, hogy a név szerinti paraméterátadásnak nincs megfelelője az Adában.

5.1.3. Mutató paraméter

Egy alprogramnak lehet *mutató paramétere* is. Ilyenkor a paraméter típusa egy névtelen általános mutató típus, ami az alprogram hívásakor jön létre.

```
procedure A ( P: access T )
```

[9]Példaként álljon itt egy program Java nyelven:

```
import java.io.*;
public class ÉrtékSzerint {
  public static void E ( int v ) {
    v++;
    System.out.println(v);
  }
  public static void main ( String args[] ) {
    E(42);
  }
}
```

A mutató paraméterek mindig `in` módúak, továbbá – mivel a mutató típusok másolat átadásúak – mindig másolat szerint adódnak át.

Egy mutató paraméter nem ekvivalens egy olyan `in` módú paraméterrel, aminek a típusa mutató a mutató paraméterben megjelölt típusra.

```
type Mut_T is access T;
procedure B ( P: in Mut_T )
```

Egy (formális) mutató paraméternek minden olyan aktuális paraméter megfeleltethető, aminek típusa konvertálható a specifikált névtelen mutató típusra. Az alábbi példában mutatunk olyan eljáráshívásokat, amelyekben az aktuális paraméter típusa mutató a formális mutató paraméterben megjelölt típusra.

```
P_1: Mut_T;
type Mut_T_2 is access T;
P_2: Mut_T_2;
V: aliased T;
...
A(P_1);         -- helyes          B(P_1);         -- helyes
A(P_2);         -- helyes          B(P_2);         -- helytelen
A(V'Access);    -- helyes          B(V'Access);    -- helytelen
```

A `V` változónak `aliased`-nek kell lennie, hogy az `Access` attribútumot alkalmazni lehessen rá. (3.6.)

A mutató paraméterek használatával hasonló eredményt érhetünk el, mint a megjelölt típusú `in out` módú paraméterek használatával:

```
procedure C ( W: in out T ) is ...
...
C(V);           -- ez a két eljáráshívás hasonló lehetőséget
A(V'Access);    -- ad a V változó manipulálására
```

Mind az `A`, mind a `C` eljárás olvashatja és írhatja a `V` változó értékét, illetve ha az `A(P_1);` és a `C(P_1.all);` hívásokat tekintjük, a `P_1` által mutatott értéket.

Az `in out` módú paraméterekkel való rokonság különösen igaz azokra a típusokra, amik hivatkozás szerint kerülnek átadásra (például a jelölt típusokra), hiszen az `A` és `C` eljárások ebben az esetben mindketten magán a `V` változóval elérhető objektumon operálnak.

```
type T is tagged record ... end record;
```

Azonban még ilyenkor is van jó néhány eltérés a kétféle paraméterezés között.[10]

[10] A különbségek abból adódnak, hogy a két esetben más a paraméterátadás során közvetített objektum-elérhetőségi információ. Lásd [2]) Egy másik példát találhatunk a függvények paraméterezésével kapcsolatban [3] 4.4.1-es szakaszában.

5.1.4. Az aktuális paraméter megfeleltetése a formálisnak

Az alprogramok hívásakor az aktuális paraméterek megadására két lehetőségünk is van. A két módszert az alábbi, az Ada.Text_IO csomagban (lásd A.5.4.) definiált eljáráson mutatjuk be:

```
procedure Get_Line ( File : in  File_Type;
                     Item : out String;
                     Last : out Natural ) ...
```

Az egyik, a hagyományos módszer szerint *pozícionális formában* kell felsorolni az aktuális paramétereket, abban a sorrendben, ahogy a formális paraméterek voltak az alprogram specifikációjában.

```
Get_Line(F,S,N);
```

A másik módszer, amikor *névvel jelölt formában* adjuk meg a paramétereket, így kapcsolva össze az aktuálisakat a formálisakkal. Ilyenkor a paraméterek sorrendje tetszőleges:

```
Get_Line(File=>F,Last=>L,Item=>S);
```

A két megfeleltetési formát lehet *keverni* is, ilyenkor mindig a pozícionálisan megadott paramétereknek kell elöl állniuk:

```
Get_Line(F,Last=>L,Item=>S);
```

5.1.5. Paraméterek feltételezett értéke

Az in módú paraméterekhez rendelhetünk feltételezett bemenő értéket. Az alprogram specifikációs része tartalmazza ezt a *feltételezett értéket*, amit akkor vesz fel a formális paraméter, ha neki megfelelő aktuális paramétert nem adunk meg. Megint az Ada.Text_IO csomagból veszünk egy példát:

```
procedure New_Line ( File    : in File_Type;
                     Spacing : in Positive_Count := 1 ) ...
```

Ezt az eljárást hívhatjuk úgy, hogy megadjuk a Spacing értékét, de úgy is, hogy nem: ilyenkor a feltételezett értéket használja az eljárás.

```
New_Line(F);       -- sort emel az F fájlban (1 sort)
New_Line(F,1);     -- az előzővel egyenértékű hívás
New_Line(F,42);    -- 42 sort emel
```

Egy alprogramnak lehet több alapértelmezett értékkel rendelkező paramétere is. Ilyenkor bármelyik aktuális paramétert el lehet hagyni, akár többet is. Ebben a paraméterek megfeleltetéséhez használt névvel jelölt forma nyújt segítséget.

```
procedure Egyenest_Rajzol ( Kezdôpont, Végpont: in Pont;
                            Vastagság: in Positive := 1;
                            Szín: in Színskála := Fekete ) ...
...
Egyenest_Rajzol(P1,P2);
Egyenest_Rajzol(P1,P2,3);
Egyenest_Rajzol(P1,P2,Szín=>Piros);
```

(Megjegyzés: Az alprogramok paramétereit általában két osztályba sorolhatjuk. Az egyikbe azok a paraméterek tartoznak, amelyeken az alprogram a fő tevékenységét végzi, vagy annak eredményét adja vissza; a másikba azok a paraméterek, amelyek ehhez a fő funkcióhoz kiegészítő adatként szolgálnak. Például a fenti, egyenes szakaszt rajzoló eljárás alapvető paramétere a szakasz kezdő- és végpontja, míg kiegészítő paramétere a vonal vastagsága és színe. Az ilyen kiegészítő paraméterekhez adunk meg általában feltételezett értéket. A feltételezett paraméterértékek bevezetése kényelmesebbé teszi az alprogramok használatát, és általánosabb[11] alprogramok írását is segíti.)

Ha egy paraméter-specifikációban több paraméter és alapértelmezett érték is van, akkor híváskor az alapértelmezett érték annyiszor kiértékelődik, ahány paraméterhez tartozik. Ebben a példában az `Él` eljárás paraméter nélküli meghívásakor két `Pont` típusú objektum jön létre, nem egy:

```
procedure Él ( A, B: in Pont_Mutató := new Pont ) is ...
...
Él;
```

5.2. Alprogram definiálása

Egy alprogram megadása háromféleképp történhet: az alprogram deklarációjának leírásával, az alprogram törzsének leírásával vagy átnevezéssel. Az első csak az alprogram specifikációját (azaz interfészét) tartalmazza. A specifikációban meg kell adni az alprogram nevét, paraméterlistáját, és függvény esetén a visszatérési értékét. Egy alprogram deklaráció:

```
function Maximum ( V: Vektor ) return Integer;
```

Az alprogram törzsének leírása esetén a specifikációtól az `is` kulcsszóval elválasztva meg kell adnunk a törzs deklarációs részét, ami az alprogram lokális változóit tartalmazza, valamint az utasításrészét, ami az alprogram hatását definiáló utasításokat tartalmazza.[12] Ez a példa bemutatja, hogyan lehet (közvetlen) rekurziót írni:

[11] Ismét – tágabb értelemben vett – polimorfizmusról beszélhetünk.

[12] Az alprogram törzsének utasításrésze egy kivételkezelt utasítássorozat, lásd 6. fejezet.

5.7. Példa
```
function Determináns (X: Mátrix) return Real is
  D: Real := 0.0;
  B: array (X'Range(2)) of Boolean := (others=>False);
  procedure Szál (P: in Real; E: in Integer; I: in Integer) is
    F: Integer;
  begin
    if I > X'Last(1) then
      D := D + P * Real((-1)**E);
    elsif P /= 0.0 then
      F := 0;
      for J in reverse X'Range(2) loop
        if B(J) then
          F := F + 1;
        else
          B(J) := True;
          Szál(P*X(i,j), E+F, I+1);
          B(J) := False;
        end if;
      end loop;
    end if;
  end Szál;
begin
  Szál(1.0,0,X'First(1));
  return D;
end Determináns;
```

Az alprogram deklarációjának az a szerepe, hogy deklarálja az alprogram szignatúráját, ami a programszöveg további részében már felhasználható az alprogram aktivizálására. Ez igen fontos akkor, ha a deklarációt és a törzset különböző fordítási egységben, vagy egy csomag (lásd 7.) specifikációjában, illetve törzsében szeretnénk elhelyezni, amikor egy alprogram törzsét az alprogram felhasználásától függetlenül (akár utána) akarjuk lefordítani. Mondunk egy példát (az általunk használt – gnat 3.09 – fordító konvencióihoz igazodva). Írjuk az a.ads fájlba a

```
procedure A;
```

az a.adb fájlba a

```
with Text_IO;
procedure A is
begin
  Put_Line("A meghívva.");
end A;
```

és az ap.ads fájlba a

```
with A;
procedure AP is
begin
  A;
end AP;
```

sorokat. Ebből a három fájlból készíthetünk úgy futtatható programot, hogy először az a.ads, aztán az ap.ads, végül az a.adb fájlt fordítjuk le, és ezután szerkesztjük össze az A-t és az AP-t. Ha pedig az A törzsén módosítani kell (de csak a törzsön, a deklaráción nem), az AP eljárást nem kell újrafordítani.

Egy másik példa az alprogram deklarációk használatára, amikor két eljárás egymást hívja (közvetett rekurzió):

```
procedure A;
procedure B is
begin
  ...
  A;
  ...
end B;
procedure A is
begin
  ...
  B;
  ...
end A;
```

Mint ebből a példából is látszik, egy alprogramtörzs megadása egyben deklarálja is az alprogramot: nem kötelező egy alprogramhoz deklarációt is és törzset is írni. Viszont egy alprogram deklarációjához mindig kell, hogy tartozzon egy alprogram megvalósítás (például egy alprogram törzs) is, ami definiálja az alprogram működését.[13]

A harmadik lehetőség alprogramok megadására az átnevezés. Az átnevezés a renames utasítással (8.4.) történik.[14]

```
function To_String ( I: Integer ) return String
                                renames Integer'Image;
```

Alprogram deklaráció megadása és átnevezés tetszőleges deklarációs részben szerepelhet, valamint csomagok specifikációs részén kívül alprogram törzs megadása is, ezért például egy alprogramnak nem csak lokális változói, hanem lokális alprogramjai is lehetnek. Ezeket szokták egymásba ágyazott alprogramoknak is nevezni.

[13] Az absztrakt alprogramok (lásd 12.7.) kivételt alkotnak ez alól a szabály alól: hozzájuk nem tartozik, nem is tartozhat megvalósítás.

[14] Egy belépési pontot is átnevezhetünk eljárássá. (Lásd 16.3.)

5.2.1. Megvalósítás megadásának módjai

Minden alprogram deklarációhoz tartoznia kell egy megvalósításnak. Egy megvalósítás nem csak az alprogram törzsével adható meg, hanem átnevezéssel is. Egy harmadik lehetőség az `Import` pragma használata. Segítségével valamilyen más programozási nyelven megírt alprogramokat kapcsolhatunk programunkhoz. (Lásd 17.)

Deklaráció és megvalósítás egyeztetése

Egy alprogram deklarációjához tartozó megvalósítás meg kell, hogy feleljen bizonyos követelményeknek: nem elég, ha ugyanaz az alprogramnév tartozik hozzájuk (hiszen az alprogram neve nem is azonosítja az alprogramot egyértelműen). A deklarációban és a megvalósításban szereplő specifikációs résznek – ami a nevet, a paraméterlistát és függvények esetén a visszatérési érték típusát tartalmazza – teljesen kompatíbilisnek kell lennie.

Az Adában négyféle kompatibilitás fogalom definiált, ezek közül a legszigorúbb a teljes kompatibilitás. Ennek ellenére még ez is egy viszonylag rugalmas definíció. Az alábbi deklaráció és törzs teljesen kompatíbilis:

```
procedure A ( X: in Integer := 32; Y: in Integer := 16#20# );
procedure A ( X, Y: Integer := 32 ) is ...
```

Az alprogramok átnevezésével kapcsolatban szóba kerülnek más kompatibilitási definíciók is, a mód kompatibilitás és az altípus kompatibilitás. (Lásd [2]) Például az alábbi átnevezés helyes:

```
procedure A ( V: in Integer := 2 );
procedure B ( V: in Integer := 3 ) renames A;
```

A teljes kompatibilitás többek között az `in` módú paraméterek feltételezett bemenő értékére is egyezést követelne meg.

5.2.2. További, az alprogramok közé sorolt entitások

Kicsit talán meglepő, de az alprogramok közé soroljuk a felsorolási típusok literáljait is. Ez a döntés megindokolja azt is, hogy a felsorolási típusú értékek nevei is átlapolhatóak. (Lásd 3.3.2. és 5.3.) A `Boolean` típusú `False` literál egy paraméter nélküli (konstans) függvényként értelmezhető.

```
type Szavak is (Citrom, Ízû, Banán, False);
... Szavak'(False) ...   -- ez egy másik függvény
```

Alprogramok továbbá az attribútumok (például az `Integer'Image` egy olyan függvény, ami `Integer`-ekhez `String`-eket rendel, lásd 3.3.1.), és az operátorok (például az `Integer`-eken értelmezett `"+"` művelet, lásd 3.1.) is. Az utóbbiak infix alakban (is) írható függvények: az `A+42` és a `"+"(A,42)` kifejezések – függvényhívások – ekvivalensek.

A felsorolási típusok literáljaihoz, az attribútumokhoz és a predefinit operátorokhoz implicit módon, a fordító által rendelődik törzs.

5.3. Alprogramok átlapolása

Korábban már láttuk, hogy a felsorolási típusok literáljai átlapolhatók, azaz több típusnak is lehet ugyanolyan nevű értéke. Ekkor egy ilyen literál esetén az, hogy melyik típus értékéről van szó, a szövegkörnyezetből deríthető ki. Az alprogramok esetében hasonló dologra van lehetőség. Érvényben lehet egyszerre például a következő két alprogram:

```
procedure Beállít ( B: in Boolean );
procedure Beállít ( Sz: in Szavak );
```

Ekkor különböző alprogramok aktivizálását jelentik a `Beállít(True);` és a `Beállít(Citrom);` eljáráshívások. A `Beállít(False);` hívás hibás, hiszen nem lehet belőle eldönteni, hogy melyik eljárást szeretnénk meghívni. Ilyen esetben a paraméter minősítésével vagy név szerinti paraméter-összekapcsolással szüntethetjük meg a kétértelműséget:[15]

```
Beállít(Szavak'(False));
Beállít(Sz=>False);
```

Összefoglalva, egy alprogramot nem csak a neve azonosít, hanem a paraméterlistája és a visszatérési értéke is (ha van). Két alprogram definiálásához használható ugyanaz az alprogramnév, de valamiben el kell térnie az alprogramok specifikációjának. Ez a valami lehet a paraméterek száma, típusa, a visszatérési érték típusa, vagy az, hogy az egyik eljárás, a másik pedig függvény.

```
procedure A;
procedure A ( I: in out Integer );
procedure A ( S: String );
function  A return Integer;
```

[15]Itt most meg sem említettük azt az esetet, amikor a szóban forgó két alprogram különböző specifikációs részben lett deklarálva. Ilyenkor ugyanis a híváskor a teljes (minősített) név kiírásával is egyértelművé tehető, melyik alprogramot kell meghívni: `Szavak_Modul.Beállít(False);`

Nem elég, ha csak egy paraméter nevében, módjában, altípusában vagy feltételezett értékében van különbség: a következő specifikáció nem különbözik eléggé a másodiktól.[16]

```
procedure A ( V: Natural := 42 );
```

Alprogramhíváskor egyértelműnek kell lennie, hogy az átlapolt alprogramok közül melyiknek kell végrehajtódnia. Nem használható az egyértelművé tevésben a kifejezések és változók közötti különbség, és az, hogy egy `out` vagy egy `in out` módú formális paraméternek csak egy változó feleltethető meg aktuálisként:

```
procedure A ( X: in Integer; Y: in Boolean := True );
procedure A ( X: in out Integer );
...
A(42);  -- nem tekintjük egyértelmûnek ezt a hívást
```

5.3.1. Operátorok átlapolása

Az alprogramok neve lehet azonosító, mint korábban láttuk, de felhasználhatjuk a nyelv operátorait is függvényeink neveként. Például definiálhatjuk a vektorok közötti szorzást a `"*"` névvel:

5.8. Példa

```
function "*" ( A, B: Vektor ) return Real is
  S: Real := 0.0;
begin
  for I in A'Range loop
    S := S + A(I)*B(I);
  end loop;
  return S;
end "*";
```

Ez a `"*"` művelet átlapolását jelenti. Ezután a vektorok közötti szorzás is írható infix alakban is: `R := P * Q;` (Az egészek, valósak, stb. közötti szorzás műveleti jele továbbra is a `"*"` marad.) Ha a `"*"` szerepel valahol a programban, és nem volna egyértelmű, hogy melyik `"*"` függvényről van szó, akkor minősítést is használhatunk:

```
R := Vektor'(P) * Vektor'(Q);
S := Komplex_Számok."*"(X,Y);
```

Egy másik példát is mutatunk. Tegyük fel, hogy definiáltuk a következő típust:

[16] Legalábbis az általunk használt fordító, a gnat 3.09 ezt mutatja...

```
type Irány is (É, ÉK, K, DK, D, DNy,Ny, ÉNy);
```

A relációs operátorok (`=`, `/=`, `<`, `<=`, `>`, `>=`, `in`, `not in`) és a diszkrét típus attribútumai automatikusan biztosítottak a típus számára. Ha további műveletekre is szükségünk van, azoknak a programjait magunknak kell megírnunk. Tegyük fel, hogy szükségünk van egy additív műveletre, ahol `D + M` azt az irányt jelenti, amelyet úgy kapunk, hogy `D`-hez hozzáadunk `M` nyolcadot az óra járásának megfelelő irányban. Ez a művelet így definiálható:

```
function "+" ( D: Irány; M: Integer ) return Irány is
begin
  return Irány'Val( (Irány'Pos(D)+M) mod 8 );
end "+";
```

Természetesen az operátor szimbólum megszokott tulajdonságai (kommutativitás, asszociativitás) átlapolás után nem teljesülnek automatikusan: ha a fenti `"+"` művelet argumentumainak felcserélhetőségét meg akarjuk őrizni, nekünk magunknak kell definiálni és megvalósítani egy ilyen műveletet:

```
function "+" ( M: Integer; D: Irány ) return Irány is ...
```

Veszélyes lehet, ha egy kifejezésen belül többféle, ugyanolyan műveleti jelű operátor is van. Ha például a `"&"` műveletet konkatenációként definiáljuk egy `Sorozat` típusú paraméteren és egy `Elem` típusún, ahol az `Elem` típus épp a `String`, akkor kérdéses, hogy mit jelent az alábbi kifejezés:

```
Str_Sorozat := Str_Sorozat & "meggy" & "leves";
```

Ilyen esetben célszerű zárójelezést használni, ez növeli a programszöveg érthetőségét.

Fontos megkötés, hogy az `in`, `not in`, `and then` és `or else` relációk nem átlapolhatók.

Az egyenlőség operátor: Az `=` és `/=` operátorok is átlapolhatók, úgy, mint a legtöbb relációs operátor.[17] Ez hasznos lehet, amikor például egy háromértékű logikát szeretnénk használni, ahol a `True` és `False` értékeken kívül van egy `Unknown` is. Egy másik példa, amikor azonos hosszúságú vektorokra definiálunk egy „elemenkénti” `=` operátort, melynek eredménye logikai értékekből álló vektor lesz.

Vannak azonban megszorítások is az `=` és `/=` relációs operátorokra. Ha általunk definiált `=` visszatérési értéke `Standard.Boolean`, a `/=` operátor automatikusan definiálódik, és eredménye az `=` ellentettje lesz. A `/=` operátort is felüldefiniálhatjuk, de csak logikaitól eltérő típusú visszatérési értékkel.

[17] Ada83-ban az `=` operátort csak `limited private` típusú operandusokkal lehetett átlapolni, és csak logikai eredményt adhatott vissza. A `/=` operátort sohasem lehetett átlapolni, ám mindig automatikusan az `=` eredményének ellentétét adta.

5.4. Alprogramok a könyv további fejezeteiben

A kivételkezelést bemutató részből (6.) megtudjuk, hogy az alprogramok törzsét egy *kivételkezelő résszel* is elláthatjuk. Ennek segítségével definiálhatjuk, hogy mi történjen akkor, ha az alprogram utasításainak végrehajtása során kivétel lép fel.

A sablonok kapcsán (11.) bemutatjuk, hogy lehet *alprogram sablonokat* is írni. Ezekből az alprogram sablonokból példányosítással (sablon paraméterek, például típusok megadásával) lehet olyan alprogramokat létrehozni, amelyek a sablonnak megfelelő módon – egymáshoz nagyon hasonlóan – működnek. Ugyanebben a fejezetben azt is bemutatjuk, hogy sablonokat (alprogram sablonokat és sablon csomagokat) hogyan kell alprogramokkal paraméterezni.

A típusokat (3.1.), csomagokat (7.) és az objektum-orientált programozás lehetőségeit (12.) bemutató fejezetekben szó esik arról, hogy *egy típusnak* mely alprogramok a *műveletei*. Megtudjuk azt is, hogy a típusműveletek hogyan *öröklődnek* a típusok származtatása során, hogy mi az a *dinamikus összekapcsolás*, és hogy milyen célt szolgálnak az *absztrakt alprogramok*.

A 9. fejezet bemutatja, hogy mi a kapcsolat a *fordítási egységek* és az alprogramok között, valamint, hogy mik azok az *alegységek*.

A párhuzamossággal foglalkozó (15. és 16.) fejezetekből megtudjuk, hogy más *hívható egységek* is vannak az alprogramokon kívül. Megtudjuk, mik a *belépési pontok* és a *védett alprogramok*.

Megismerkedhetünk még az *alprogramra mutató típusokkal* és használatukkal (3.6.) is. Az alprogramra mutató típusok azért fontosak, mert segítségükkel átadhatunk alprogramokat más alprogramoknak paraméterül, anélkül, hogy sablonokat kelljen használni.

Más programozási nyelveken megírt alprogramjainkat összeszerkeszthetjük Ada programokkal. Ehhez, és az *alprogram hívási konvenciók* manipulálásához adnak segítséget a `Convention`, `Import` és `Export` pragmák. (Lásd 17.)

Az `Inline` pragma használatával növelhetjük programunk hatékonyságát: lehetőség van arra, hogy egy alprogram törzse „makrószerűen" *behelyettesítődjön* a meghívása helyén.

5.5. Feladatok

5.1. Írjon – rekurzívan és rekurzió nélkül – Fibonacci-számokat előállító függvényt! Hogyan lehet úgy megírni a rekurzív változatot, hogy ne legyen sokkal kevésbé hatékony, mint a rekurzió nélküli?

5.2. Elemek sorozataként (vagy listájaként) ábrázolt halmazokhoz készítsen "*" és a "=" műveleteket a megfelelő operátorok felüldefiniálásával! Az első két halmaz metszetét számolja ki, a második eldönti, hogy két halmaz egyenlő-e.

5.3. Írjon olyan függvényt, ami egy tömbben tárolt természetes számokról eldönti, hogy relatív prímek-e!

5.4. Állapítsa meg, hogy az Ön által használt fordító másolat vagy hivatkozás szerint adja át a tömböket és a rekordokat!

5.5. Készítsen „quicksort” algoritmus alapján rendező eljárást!

6. Fejezet

Kivételek, kivételkezelés

Próbáljuk meg összefoglalni, mit is jelent a „kivételes esemény" kifejezés egy programozó számára. Kivételes esemény akkor történik, amikor egy olyan utasítást akar végrehajtani egy program, amit – az adott körülmények között – nem lehet. Gondolhatunk itt valami egyszerű, számítógép-közeli dologra, például, hogy megkísérlünk nullával osztani, vagy bonyolultabbra, például arra, hogy egy üres halmazból megpróbálunk kivenni egy elemet. Az eredmény mindkét esetben hasonló lehet: a program nem tudja végrehajtani az utasítást, és ezt például azzal jelzi, hogy „elszáll". Miért következhet be egy kivételes esemény? Lehet az az ok, hogy rossz a programunk: például bizonyos esetekre elfelejtettünk gondolni a megírása közben. Az is lehet, hogy jó a program, csak éppen rosszul használjuk: olyan körülmények között, amelyek nem teljesítik az előfeltételét. (Például az a program vagy programegység, amiben a kivételes esemény történik, jó, de az őt meghívó program nem megfelelő paramétereket ad át neki. Egy másik példa erre egy interaktív program esetén a felhasználóval való kommunikáció, melynek során gyakran előfordul, hogy a felhasználó hibás adatot gépel be.) Végül az is lehet, hogy a programot direkt úgy írtuk meg, hogy bizonyos esetekre úgy reagáljon, hogy megszakítja a működését, hogy egy „természetes" válasza legyen az, hogy egy hibaüzenettel megáll.

Programfejlesztői szempontból érdemes még egy gondolatot szánni a második esetnek. Általában nem elég, ha egy program jó: egy igazán használható programterméknek megbízhatónak is kell lennie. Ez alatt azt értjük, hogy szélsőséges, vagy akár hibás körülmények közé helyezve is úgy kell működnie, hogy ne okozzon kárt (például, ha egy hibás bemenő adatra el is „száll" a program, az addig kiszámolt kimenő adatok ne sérüljenek meg vagy vesszenek el), esetleg igyekezzen saját maga biztosítani a működéséhez szükséges feltételeket. (Például, ha a felhasználó hibás adatot gépel be, vegye észre és kérje be az adatot mégegyszer.)

Nagyon nagy munka olyan programot írni, amely megfelel ennek a követelménynek. Sok, néha végeláthatatlanul sok esetet végig kell gondolni, olyanokat, amik „elvileg be sem következhetnek", és mindre fel kell készíteni

a programot: ha ez történik, ezt csináld, ha az, akkor meg azt... Gyakran az elkészült program igen nagy hányadát a kivételes események lekezelése teszi ki. Ez a kód olvashatóságát, érthetőségét (ami szintén egy fontos összetevő egy programtermék minőségének meghatározásakor) nagy mértékben rontja. Egy igényes programozási nyelvnek erre megoldást kell nyújtania, különböző, kivételes események lekezelésére szolgáló nyelvi eszközök, mechanizmusok bevezetésével.

Mit tehet a programozó, ha egy olyan nyelvet használ, amiben nincsenek különleges, eleve erre a célra szánt nyelvi elemek? Milyen megoldásokat láthatunk például C programokban? Gyakran extremális elemet (vagy elemeket) használnak visszatérési értékként olyan függvényeknél, amelyek végrehajtása során kivételes esemény adódhat. Például normális esetben a függvényhívás pozitív számmal tér vissza, hiba esetén pedig -1 vagy -2 lesz az eredmény, a hiba fajtájától függően. Az is gyakran előfordul, hogy egy globális változó értékét kell lekérdeznünk egy művelet meghívása után, mert ez a változó jelzi, ha valami kivételes dolog történt. Mind a két technika használata mellett a függvényt hívó programegységben elágazás utasítást alkalmazhatunk a visszatérési érték illetve a globális változó tesztelésére. Ha pedig egymás után több olyan függvényt is meghívunk, ami a kivételes eseményeit, hibáit a visszatérési értékében, egy paraméterben, vagy egy globális változóban adja vissza, a kód még inkább elbonyolódik: egymásba ágyazott elágazásokat kell írnunk. Az elágazások eme szövevényében csak egy ág lesz a normális, hiba nélküli lefutást leíró utasítássorozat, a többi valamilyen (feltehetőleg nagyon ritkán előforduló) hibás eset lekezelése lesz.

6.1. Példa

```
with Egyik, Másik, Visszavon;
with Text_Io; use Text_Io;
procedure Egymás_Után ( I: in out Integer ) is
begin
  Egyik(i);
  case I is
    when -1 =>     -- Egyik: hibás paraméter
                   I := -1;
    when -2 =>     -- Egyik: kevés elem volt a még veremben
                   Visszavon;
                   I := -1;
    when others => -- Egyik: sikeres végrehajtás
                   I := Másik(i);
                   if I >= 0 and then I < Integer'Last-10 then
                     -- Másik és Egymás_Után:
                     -- sikeres végrehajtás
                     I := I+10;
                   elsif I >= Integer'Last-10 then
```

```
                        -- Másik sikerült, de Egymás_Után nem
                        Put_Line(Current_Error,
                              "Egymás_Után: Túl nagy érték!");
                        I := -2;
                      else      -- I < 0
                        -- Másik: sikertelen memóriafoglalás
                        I := -3;
                      end if;
  end case;
end Egymás_Után;
```

Mint érezzük is ezen példa után, a programozó könnyen kísértésbe eshet, hogy egyáltalán ne foglalkozzon a kivételes események lekezelésével: lustaságát szépérzéke is támogatja ebben a döntésben. Mint majd látni fogjuk, az Ada kivételfogalma és kivételkezelő mechanizmusa részben megoldja a problémát, segítségével megbízható, jó hibatűrő tulajdonsággal rendelkező, mégis elegáns, kompakt kódot írhatunk.

6.1. Kivételkezelés az Ada nyelvben

Egy Ada programban *kivétel váltódik ki*, ha egy utasítás valami miatt nem tud végrehajtódni. Ennek oka lehet például egy számolási probléma, nullával való osztás, túlcsordulás, vagy memóriacímzési, tömbindexelési hiba, esetleg memóriahiány (ezeknek mind megfelel valamilyen *előre definiált kivétel*), de a programozó maga is kiválthat, sőt, definiálhat kivételeket.

Sok más programozási nyelvben írt program egy kivétel fellépése esetén megszakítja működését: jó esetben egy futási idejű hibát jelző üzenetből, vagy nyomkövetéssel tudhatjuk meg, mi volt a probléma. Az Ada ezzel szemben megadja a programozó számára annak lehetőségét, hogy hibák és egyéb kivételes végrehajtási helyzetek esetén rendelkezzék ezek *lekezeléséről*. A kódban a kivételkezelő részek a normális végrehajtást leíró utasítássorozattal nem keverednek, ez biztosítja a kód könnyebb átláthatóságát.

A kivételeket nem feltétlenül ott szeretnénk lekezelni, ahol kiváltódnak. Különösen igaz ez a programozó által kiváltott kivételekre: a kivétel kiváltása itt éppen azt jelzi, hogy az adott kontextusban nem tudtunk semmi más értelmes dolgot tenni, a kivételes helyzet megoldását egy másik programegységre akarjuk bízni. Az Ada programban a kivételek a blokkok hívási sorrendjének megfelelően *továbbterjednek*, amennyiben nem kezeljük le azokat. Ha egyáltalán nem talál megfelelő kivételkezelőt, a program futási idejű hibával véget ér.

Az Adában[1] a *kivétel példány* fogalom használható arra, hogy megkülönböztessük a programvégrehajtás különböző pontjain kiváltódott kivételeket.

[1] Ez a fogalom az Ada95-ben jelent meg.

Itt a programvégrehajtás szempontjából különbözőnek tartjuk a programkód ugyanazon utasításának különböző végrehajtásait.

6.1.1. A kivételek fajtái

Előre definiált kivételek: A `Standard` csomagban definiált kivételeket *előre definiált* kivételeknek nevezzük. Ezek valamelyike lép fel, amikor egy utasítás végrehajtása vagy egy kifejezés kiértékelése közben egy, a nyelv által definiált ellenőrzés elbukik. Az előre definiált kivételek és kiváltódásuk okai a következők:

`Constraint_Error`: akkor léphet fel, ha egy megszorítás megsértése (például egy változó olyan értéket kap, ami kívül esik a típusértékhalmazán), indexhiba, diszkrimináns megsértése, vagy `null` értékű mutató által mutatott elemre történő hivatkozás következik be;

```
procedure CE is
  I: Natural := 100;
begin
  loop
    I := I-1;
  end loop;
end CE;
```

`Program_Error`: akkor léphet fel, ha olyan alprogram, taszk vagy sablon aktivizálására történik kísérlet, amelynek törzse még nincs kiértékelve, ha egy függvény végrehajtása befejeződött, de nem volt `return` utasítás, vagy ha egy szelektív várakoztatás (lásd 15.8.1.) minden ága zárt és nincs az utasításnak `else` ága;

```
procedure PE is
  generic procedure G;
  procedure P is new G;
  procedure G is begin null; end;
begin
  P;
end PE;
```

`Storage_Error`: akkor léphet fel, ha nem áll rendelkezésre elég memória egy igényelt allokáláshoz;

`Tasking_Error`: taszkok közötti kommunikáció során léphet fel. (Lásd 15.3.)

A `Numeric_Error` kivétel az Ada95-ben[2] a `Constraint_Error`-nak felel meg (a `renames` utasítást lásd 8.4.):

```
Numeric_Error: exception renames Constraint_Error;
```

Egyéb, a nyelv által definiált kivételek: Az előre definiált kivételek mellett van jópár kivétel, amit az előre definiált könyvtárak definiálnak, azok alprogramjai váltanak ki. Ezek közül a legfontosabbakat, leggyakrabban előfordulókat a bemenettel és kimenettel (szövegfájlok: A.5.4., szekvenciális fájlok: A.5.1., direktfájlok: A.5.2.) kapcsolatban ismerhetjük meg, mint például az `Ada.IO_Exceptions.End_Error`-t. Ebben a könyvben a matematikai könyvtáraknál (G. fejezet) is találkozhatunk még további kivételekkel.

Kivételek definiálása: A programozó saját maga is definiálhat kivételeket. Ez a szerkezet leginkább a változódeklarációkra hasonlít, de fontos megjegyezni, hogy a kivételek nem változók.

```
Hibás_Fájlnév: exception;
Üres_Verem, Teli_Verem: exception;
```

Kivételek definíciója tetszőleges deklarációs részben szerepelhet, a kivétel nevének láthatósága a megszokott láthatósági szabályoknak megfelel. Azonban a kivételek érvényességi köre túlmutat deklarációjuk hatáskörén: egy kivétel (továbbterjedése során) a program végrehajtásának egy olyan szakaszán is kiváltódhat, ami kívül esik a deklarációja hatáskörén. (Példa: 6.2.2.) A programozó által definiált kivételek természetesen automatikusan nem lépnek fel, kiváltásukról minden esetben a programozónak kell gondoskodnia.

6.1.2. A kivételkezelő rész

Egy blokkhoz, a blokkban szereplő utasítássorozathoz hozzákapcsolhatunk egy kivételkezelő részt. Ez a kivételkezelő rész használható a blokkban fellépő kivételek elfogására, lekezelésére. Kivételkezelő rész tartozhat alprogramhoz, csomag törzséhez, blokk utasításhoz, taszk törzséhez, és belépési ponthoz tartozó törzshöz (az `accept` utasítást is beleértve).[3] Az ezekben a programrészekben található utasítássorozatot közös néven *kivételkezelt utasítássorozatnak* hívjuk.

[2] Az Ada83-ban ez a kivétel még egy különálló előre definiált kivétel volt, ez bizonyos esetekben inkompatibilitást és inkonzisztenciát (lásd [2]) is jelenthet a nyelv régi és új definíciója között.

[3] Az Ada83-ban a belépési pontokhoz tartozó törzsek nem tartalmazhattak kivételkezelő részt. (Mivel védett egységek sem voltak, belépési ponthoz csak `accept` utasítások szolgálhattak törzzsel: ezekhez viszont csak egy blokk utasítás segítségével lehetett kivételkezelést kapcsolni.)

```
begin   -- vagy do
  ...   -- utasítások              --  ilyen lehet egy
exception                          --  kivételkezelt
  ...   -- kivételkezelő ágak      --  utasítássorozat
end;
```

Mint láttuk a könyv eddigi fejezeteiben, egy ilyen kivételkezelt utasítássorozat nem biztos, hogy tartalmaz kivételkezelő részt. A kivételkezelő rész, ami egy `case`-szerű szerkezet (lásd 4.2.1.), az `exception` kulcsszóval kezdődik és a blokkhoz tartozó `end`-ig tart. A kivételkezelő részen belül kivételkezelő ágak találhatóak (legalább egy), amelyek a `when` kulcsszóval kezdődnek. Egy ág több kivétel megnevezését is tartalmazhatja, valamint egy utasítássorozatot, amit akkor kell végrehajtani, amikor a blokkban valamelyik megnevezett kivétel kiváltódik.

```
exception
  when Name_Error => Put_Line("Hibás fájlnevet adott meg!");
  when End_Error | Hibás_Formátum =>
                    Close(Bemenet_fájl); Close(Kimenet_fájl);
                    Put_Line("A fájl szerkezete nem jó.");
```

Vizsgáljuk meg, hogyan történik egy kivételkezelt utasítássorozat végrehajtása. Használjuk ehhez az alábbi példát:

```
  utasítás_01;
  utasítás_02;
  utasítás_03;
exception
  kivételkezelő_ág_01;
  when kivétel_01 | kivétel_02 =>
           utasítás_11;
           utasítás_12;
           utasítás_13;
  kivételkezelő_ág_03;
  kivételkezelő_ág_04;
```

Normális, kivétel nélküli lefutás esetén a 01, 02 és 03 jelű utasítások hajtódnak végre. Ha valamelyik utasítás közben kivétel lép fel, a végrehajtás azon a ponton megszakad. Ha nincs kivételkezelő rész, akkor egyben a kivételkezelt utasítássorozat is véget ér, a hátralevő utasítások nem hajtódnak végre.[4] (Mint majd látni fogjuk, ebben az esetben a blokk abnormális módon ér véget, ami azt jelenti, hogy a hívó blokkban is fellép

[4] Ha egy eljárás utasításai közben váltódik ki kivétel, az eljárás végrehajtása megszakad, a vezérlés visszatér a hívó blokkba, és az eljárás `in out` és `out` módú paramétereinek megfelelő változók értéke definiálatlan.

a kivétel. Lásd 6.1.3.) Ha van kivételkezelő rész, de a fellépett kivétel nincs megnevezve egyik ágban sem, akkor az előző esethez hasonlóan ér véget a blokk működése. Viszont, ha van egy olyan kivételkezelő ág, amelynek egyik (vagy egyetlen) megnevezett kivétele az, ami kiváltódott (legyen ez például a második ág), a hozzá tartozó utasítássorozat hajtódik végre, a példánkban a 11, 12 és 13 jelű utasítások. Ezzel a kivétel le lett kezelve és a blokk (normálisan, nem abnormálisan, mint az előző két esetben) véget ér. Tehát, ha például a 02 jelű utasítás közben fellépett a 01 jelű kivétel, az 11, 12 és 13 utasítások fognak végrehajtódni, a 03 jelű nem.

Van néhány szabály, ami további megszorításokat tesz a kivételkezelő részre.

- Egy kivétel többször is szerepelhet egy ágban. Tegyük fel, hogy a `My_Direct_IO`-beli `End_Error`, ugyanúgy, mint az `Ada.Text_IO`-beli, az `Ada.IO_Exceptions.End_Error` átnevezése.[5] (A `Numeric_Error`-ról pedig már korábban elárultuk, hogy a `Constraint_Error`-nak felel meg.[6])

  ```
  when Text_IO.End_Error | My_Direct_IO.End_Error => ...
  when Constraint_Error | Numeric_Error => ...
  ```

 Ennek a szabálynak különösen dokumentációs szempontból van haszna, segíthet a kód olvashatóbbá tételében.

- Egy kivétel nem szerepelhet több ágban.[7]

  ```
  when End_Error | Constraint_Error => ...
  when Numeric_Error => ...   -- ez hibás Ada95-ben
  ```

- A kivételkezelő rész tartalmazhat `others` ágat is, de az `others` csak az utolsó ágban lehet, és csak magában.

  ```
  exception
    when Constraint_Error => null;
    when others => Put_Line("Nem Constraint_Error.");
  end;
  ```

 Az `others` ág minden olyan kivételt elfog, amit a fölötte levő ágak nem fogtak el; ez alól még azok a kivételek sem kivételek, amelyek az adott kontextusban nem láthatóak. (Lásd még: 6.2.2.)

[5] Például a My_Direct_IO az Ada.Direct_IO példányosításaként jött létre.

[6] Mint már említettük, Ada83-ban a két kivétel még különbözött.

[7] Ez a példa egyben egy olyan érdekes esetet is szemléltet, amikor egy helyes Ada83 program helytelen Ada95-ben: Ada83-ban az ütközést kiváltó két kivétel még különböző volt.

Sokszor hasznos lehet, hogy egy kivételkezelő ágban hozzáférjünk az elfogott kivétel nevéhez, vagy egyéb információkhoz. Ehhez nyújtanak segítséget a *kiválasztó paraméterek*. Használatukkal ilyen kivételkezelő ágakat is írhatunk:

```
when Kivétel: End_Error | Formátum_Hiba => <utasítássorozat>
```

A kiválasztó paraméterekről részletesebben 6.1.5. fog foglalkozni.

6.1.3. Kivételek kiváltódása, terjedése

Keletkezésük helye szerint három csoportba oszthatjuk a kivételeket. Egy deklarációs részben, egy kivételkezelő részben vagy egy (a kivételkezelő részen kívül elhelyezkedő) utasítássorozat részben léphet fel kivétel. Ahhoz, hogy megértsük, a kivételkezelő keresése melyik esetben hol kezdődik, meg kell ismerkednünk a *dinamikus tartalmazás* fogalmával.

Dinamikus tartalmazás: Ha egy "A" programrész végrehajtása magában foglalja egy "B" programrész végrehajtását, akkor azt mondjuk, hogy amíg a "B" programrészt végrehajtjuk, az "A" dinamikusan tartalmazza a "B"-t. (A dinamikus tartalmazás irreflexív, antiszimmetrikus és tranzitív.) A legbelső dinamikusan tartalmazó programrész az a dinamikusan tartalmazó programrész, amely időben legkésőbb kezdődött. (Úgy is fogalmazhatunk, hogy a legbelső dinamikusan tartalmazó programrész az, amelyik a szóban forgó programrész végrehajtását kezdeményezte.)

```
procedure A is
  procedure B is ... begin ... end B;
begin
  declare
    procedure C is ... begin ... B; ... end C;
  begin
    C;
    B;
    ...
  end;
  ...
end A;
```

Például a C eljárás végrehajtásakor a blokk utasítás dinamikusan tartalmazza C-t, az A pedig dinamikusan tartalmazza a blokk utasítást. A B eljárás első végrehajtásának ideje alatt C dinamikusan tartalmazza B-t, de a második végrehajtása alatt természetesen nem: ekkor a blokk utasítás dinamikusan tartalmazza B-t, de már nem tartalmazza dinamikusan C-t.[8]

[8]Egy programegység egy végrehajtási példánya tartalmazhatja dinamikusan ugyanazon programegység egy másik végrehajtási példányát: gondoljunk csak a rekurzióra.

Egy kivétel kiváltódásánál a kivételkezelőt általában a legbelső dinamikusan tartalmazó kivételkezelt utasítássorozatban kell kezdeni keresni. (Mint már említettük, a kivételkezelt utasítássorozat lehet blokk utasítás, valamint alprogram, csomag, taszk törzse és belépési ponthoz tartozó törzs, például `accept` utasítás.) Ha ott nincs megfelelő kivételkezelő, a keresés az ezt dinamikusan befoglaló legbelső kivételkezelt utasítássorozatban folytatódik. Ha ez, a hívási láncban kifelé haladó keresés sehol sem jár eredménnyel, akkor kijutunk a főprogramba. (Tegyük fel, hogy a programunkban nincsenek más taszkok – lásd 15.) A főprogramnak – ami egy eljárás – a törzse a legkülső kivételkezelt utasítássorozat. Ha itt sincs megfelelő kivételkezelő, a program futási hibával véget ér.

A három keletkezési hely: Térjünk vissza a példánkhoz. Ha a kivétel egy deklarációs részben keletkezik, például a C deklarációs részében, akkor a kivételkezelőt a C-t hívó kivételkezelt utasítássorozatban, a blokk utasításban kell kezdeni keresni. Amennyiben itt nincs megfelelő kivételkezelő, a blokk utasítás végrehajtása abbamarad (a B nem hívódik meg), és a kivétel továbbterjed, fellép a blokk utasítást aktiváló A eljárás törzsében (még a kivételkezelő rész előtt).

Az A törzsének végrehajtása a hozzá tartozó kivételkezelő rész tesztelésével folytatódik. Ha ez sem tartalmaz a szóban forgó kivételt megnevező ágat, az A eljárás végrehajtása megszakad, és a kivétel fellép az A-t meghívó blokkban, vagy, ha az A a legkülső blokk, a főprogram, a program hibaüzenettel leáll.

A harmadik eset, amikor a kivétel egy kivételkezelő részben lép fel: miközben épp egy kivételt kezelünk le mondjuk a C-ben, a lekezelő ághoz tartozó utasítássorozat egyik utasítása kivétel kiváltásával ér véget. Ebben az esetben a fellépő kivételhez nem a C kivételkezelő részében kell kezdeni a kivételkezelő keresését, hanem eggyel kijjebb, a blokk utasításban.

6.1.4. Kivételek kiváltása, terjesztése

Egy kivétel kiváltását a programozó is kezdeményezheti a `raise` utasítás használatával. Amint a példákból is látjuk, tetszőleges kivételt kiválthatunk vele, mind előre definiáltat, mind előre definiált csomagban, illetve programozó által definiáltat:

```
raise Constraint_Error;
raise Ada.IO_Exceptions.End_Error;
raise Verem_Csomag.Üres_Verem;
```

A `raise` utasítás hatására az előző szakaszban megismert terjedési mechanizmus lép érvénybe, éppúgy, mintha például egy `Constraint_Error` váltódott volna ki egy hibás tömbindex hatására: ha a `raise` utasítás egy

K kivételkezelő részben van, kivételkezelőt abban a legbelső kivételkezelt utasítássorozatban kell kezdeni keresni, ami a szóban forgó K kivételkezelő részt magában foglaló kivételkezelt utasítássorozatot dinamikusan tartalmazza. Ha egy kivételkezelő részen kívül helyezkedik el a `raise` utasítás, akkor a tartalmazó kivételkezelt utasítássorozat kivételkezelő része (ha van) az elsődlegesen illetékes kivételkezelő rész.

Egy kivételkezelő ághoz tartozó utasítássorozatban a `raise` utasítás állhat magában, paraméter nélkül is. Ilyenkor annak a kivételnek (kivétel példánynak, 6.1.5.) az újrakiváltását eredményezi, amelyiket a kivételkezelő ág éppen lekezel:

```
procedure Beolvas is
   ...
begin
   ...
exception
   when End_Error | Constraint_Error => Close(f); raise;
   when Hibás_Fájlnév => Put_Line("Beolvas: Rossz fájlnév!");
                         raise;
   when others =>
      Put_Line(Current_Error,"Beolvas: Ismeretlen hiba!");
      raise;
end Beolvas;
```

Ha a kivételkezelő ág több kivételt is megnevez (például `others`-ág esetén), akkor természetesen az a kivétel kerül újrakiváltásra, ami aktivizálta az ágat.

6.1.5. Kivétel példányok

Sokszor hasznos lehet, ha egy kivételkezelő ágban a fellépett kivételről több információval is rendelkezünk, mint hogy a kivétel valamelyike az ág elején megnevezetteknek. Lehet, hogy két különböző kivételhez csak egy kis részletben eltérő kivételkezelőt szeretnénk írni: ilyenkor jól jöhet, ha nem kell a két kivételhez két különböző ágat kapcsolni. Erre (is) kínál lehetőséget a kivétel példány fogalma, és a már emlegetett kiválasztó paraméter.[9]

A kivétel példány egy kivétel (például a `Constraint_Error`) egyszeri kiváltódásának, fellépésének feleltethető meg. A `Constraint_Error` kivétel a program futása során többször is kiváltódhat, akár a kód ugyanazon sorában is: minden kiváltódásakor egy új kivétel példány jön létre. Egy kivétel példány mindig az `Ada.Exceptions.Exception_Occurrence` típusba tartozik, függetlenül attól, hogy melyik kivétel példánya.

[9] Ezek a nyelvi elemek csak az Ada95-ben kerültek bevezetésre.

A kiválasztó paraméter egy Exception_Occurrence típusú konstans (azaz egy kivétel példány), vele hivatkozhatunk egy kivételkezelő ágon belül az éppen kezelt kivétel példányra:

```
when E: End_Error | Data_Error => Put_Line(Exception_Name(E));
```

Ebben a példában más és más kerül kiírásra, ha End_Error vagy ha Data_Error váltódott ki a kivételkezelő által védett utasítássorozatban.

Az Ada.Exceptions csomag

Az Exception_Occurrence típus az Ada.Exceptions csomagban van definiálva. A következőkben áttekintjük majd a fontosabb műveleteit, de előtte meg kell ismerkednünk a szintén itt definiált Exception_Id típussal. Ez a típus minden kivételhez egy egyedi azonosítót rendel, amit az Identity attribútummal kaphatunk meg, például: Constraint_Error'Identity. Az Exception_Identity függvény megadja, hogy egy kivétel példány melyik kivételnek a példánya:

```
function Exception_Identity(X : Exception_Occurrence)
                                    return Exception_Id;
```

Az Exception_Name függvények – akár egy Exception_Id, akár egy Exception_Occurrence típusú paraméterrel hívjuk meg – a kiváltott kivétel teljes nevét adják meg. Az előző példánál maradva, a

```
raise Text_IO.Data_Error;
```

utasítás hatására ADA.IO_EXCEPTIONS.DATA_ERROR fog kiíródni. Egy kivétel példányról rövid, egysoros ismertetőt kaphatunk az Exception_Message függvény használatával. Ezt az ismertetőt, mint majd látni fogjuk, a kivétel példány kiváltásakor meg is lehet adni, de természetesen ez nem kötelező. Az előre definiált ismertetők szövege a fordítóprogramtól is függhet. Ezek általában nem tartalmazzák a kivétel nevét. Ezzel szemben a sztring, amit az Exception_Information függvény visszaad (és ami megint csak függ a fordítóprogramtól is) a kivétel példányhoz tartozó nevet is és ismertetőt is szokta tartalmazni, és még további információval is szolgálhat, például a kiváltás pillanatában érvényes hívási láncról.

A Raise_Exception eljárás, hasonlóan a raise utasításhoz, egy kivétel kiváltására használható. Hatására létrejön egy új kivétel példány, amelyhez tartozó ismertető (amit az Exception_Message visszaad) az lesz, amit az eljárás második paraméterében megadunk:

```
raise Constraint_Error;
Raise_Exception(Constraint_Error'Identity);
Raise_Exception(Constraint_Error'Identity,
     "A 'Pontok' tömb utolsó eleme nem egyenlő az elsővel!");
```

Mindhárom utasítás `Constraint_Error` kivételt fog kiváltani, de a létrejövő kivétel példányokhoz más-más ismertető tartozhat (a `raise` utasítás esetén egy alapértelmezett szöveg, a második esetén az üres, a harmadik esetén pedig a megadott `String`).

A `Reraise_Occurrence` eljárással egy kivétel példányt újra ki tudunk váltani, hasonlóan a magában álló, paraméter nélküli `raise` utasításhoz: hatásukra nem jön létre új kivétel példány.

```
when E: others => Reraise_Occurrence(E);  -- raise;
```

Ezzel az eljárással viszont nem csak egy kivételkezelő ágon belül találkozhatunk: ha van egy érvényes tartalommal rendelkező `Exception_Occurrence` típusú változónk, a kivétel példányt a programban bárhol újra kiválthatjuk.

Egy ilyen érvényes `Exception_Occurrence` értéket változóban eltárolni a `Save_Occurrence` eljárás és függvény segítségével lehet. Az alábbi programrészlet egy nagyon bonyolult megvalósítása a `raise Constraint_Error` utasításnak:

```
declare
  E: Exception_Occurrence;
begin
  begin
    raise Constraint_Error;
  exception
    when CE: Constraint_Error => Save_Occurrence(E,CE);
  end;
  Reraise_Occurrence(E);
end;
```

6.2. Példák kivételek használatára

Az alábbiakban megpróbáljuk bemutatni (és példákkal illusztrálni), hogy a kivételeket mi mindenre lehet használni.

6.2.1. Mire használhatjuk a kivételeket?

Hibajelzés: Használhatjuk a kivételeket hibajelzések adására: ha a program adathibát észlel, kivételt vált ki, ezzel jelezve az őt aktivizáló blokknak a számítás sikertelenségét. Az alábbi példában a `Sor` típus két művelete egy-egy hibával jelzi, ha nem tudja a kért transzformációt elvégezni.

6.2. Példa

```
...
package Sor_Csomag is
  type Sor is limited private;
  ...
  procedure Betesz( S: in out Sor; E: in Elem );
  procedure Kivesz( S: in out Sor; E: out Elem );
  Tele_Sor, Üres_Sor: exception;
private
  type Vektor is array (0..Méret-1) of Elem;
  type Sor is record
    Betéve, Kivéve: Natural := 0;
    Adat: Vektor;
  end record;
end Sor_Csomag;

package body Sor_Csomag is
  ...
  procedure Betesz( S: in out Sor; E: in Elem ) is
  begin
    if S.Betéve=S.Kivéve+Méret then raise Tele_Sor; end if;
    S.Adat(S.Betéve mod Méret) := E;
    S.Betéve := S.Betéve+1;
  end Betesz;

  procedure Kivesz( S: in out Sor; E: out Elem ) is
  begin
    if S.Kivéve=S.Betéve then raise Üres_Sor; end if;
    E := S.Adat(S.Kivéve mod Méret);
    S.Kivéve := S.Kivéve+1;
    if S.Kivéve = Méret then
      S.Kivéve := S.Kivéve - Méret;
      S.Betéve := S.Betéve - Méret;
    end if;
  end Kivesz;
end Sor_Csomag;
```

Biztonságos menekülés: A következő kódrészlet a kivétel felismerése után még elvégez néhány befejező műveletet, és csak azután jelzi a hibát a hívó szintjén. Ezzel a stratégiával biztosíthatjuk, hogy a hibásan működő programrész valamennyire konzisztens adatszerkezetet hagyjon maga után.

```
begin
  ... -- adatok kiírása az F fájlba
```

```
exception
  when Constraint_Error => Close(F); raise;
end;
```

Ugyanezt a technikát használhatjuk a programfejlesztés tesztelési fázisában. Meghatározhatjuk, hogy egy hiba a program végrehajtás melyik szakaszában lép fel, esetleg fontosabb lokális változók értékét is kiírathatjuk a képernyőre:

```
when E: others =>
    Put_Line("Hiba a Feldolgoz eljárásban! DbSzam = " &
             Integer'Image(DbSzam) &
             Exception_Name(E) & Exception_Message(E));
```

Megmentés: A kivételkezelő program kialakítható úgy, hogy felismerje a hibát, korrigálja azt, és a programegység vagy blokk helyett adjon egy többé-kevésbé használható eredményt, megmentve a programot a leállástól. Jó példa erre az a szintaktikus elemző, mely, ha az elemzés egy szakaszán nem ismeri fel a bemenetet, azt a szakaszt átugorva, egy későbbi, már felismert, helyes szakaszon folytatja működését.

Egy másik példa lehet a fentebb bemutatott Sor típust és műveleteit interaktívan kipróbáló program: ha egy műveletet nem sikerül végrehajtani a soron, a program figyelmen kívül hagyja a műveletre való felszólítást.

```
...
  S: Sor;
  E, Ee: Integer := 1;
  C: Character;
begin
  loop
    Get(C); Skip_Line;
    exit when C='v';
    begin
      if C = 'b' then Betesz(S,E); E:=E+1;
                 else Kivesz(S,Ee);
      end if;
      Kiir(S);
    exception
      when E: others => Put_Line(Exception_Name(E));
    end;
  end loop;
...
```

Vezérlési szerkezet: Elvileg az is előfordulhat, hogy a kivételkezelést vezérlési szerkezetként szeretnénk használni. Alkalmas például arra, hogy

segítségével gyorsan és egyszerűen kilépjünk többszörösen egymásba ágyazott ciklusokból. Azt azonban nem szabad elfelejtenünk, hogy a kivételkezelő mechanizmus általában nagyon erőforrás-igényes, programunk hely- és időigényét jelentősen növelni tudja. Ezért ilyen esetekben igyekezzünk más megoldást keresni: egymásba ágyazott ciklusokból való kilépésre a `exit` utasítás is használható...

Ez a programegység bemutatja, hogyan lehet ciklus és elágazás nélkül, pusztán rekurziót és kivételkezelést alkalmazva faktoriálist számító függvényt írni:

6.3. Példa
```
function Faktoriális ( N: Natural ) return Natural is
begin
  return Faktoriális(N-1) * N;
exception
  when Constraint_Error => return 1;    --  N = 0
end Faktoriális;
```

6.2.2. További példák

Az alábbi példa azt szemlélteti, amikor egy kivételkezelő ágban elhelyezett `raise` utasítás nem újra kivált egy kivételt, hanem egy fellépett kivételt helyettesít egy másikkal:

6.4. Példa
```
function Keres ( L: Lista ) return Lista is
  P: Lista := L;
begin
  while P.Tartalom /= 42 loop
    P := P.Következô;
  end loop;
exception
  when Constraint_Error => raise Nincs_42;
end Keres;
```

Ebben a példában megmutatjuk, hogy hogyan alakul egy kivétel deklarációjának láthatósági, hatás- és érvényességi köre.

6.5. Példa
```
procedure A is
  Hiba1, Hiba2: exception;
begin
  declare
      Hiba2, Hiba3: exception;
```

```
  begin
    null;
  exception
    when Hiba2 => ...
    when A.Hiba2 => ...
  end;
exception
  when Hiba3 => ...    -- helytelen, fordítási hibát okozó sor
  when others => ...   -- lekezel minden hibát, pl. Hiba3-at is
end A;
```

Ez a példa azt szemlélteti, hogy miben különbözik a kivételek kiváltása a kivételek újra kiváltásától: az előbbi esetén új kivétel példány jön létre, az utóbbi esetén egy, már meglévő kivétel példány terjed tovább.[10]

6.6. Példa

```
with Text_IO, Ada.Exceptions; use Text_IO, Ada.Exceptions;
procedure Újra is
  Hiba: exception;
  O: Exception_Occurrence;
begin
  begin
    Raise_Exception(Hiba'Identity,"Na ugye.");
  exception
    when E: Hiba => Save_Occurrence(O,E);
                    Put_Line(Exception_Message(O));
                    raise; -- raise Hiba;
  end;
  Put_Line(Exception_Message(O));
exception
  when E: Hiba => Put_Line(Exception_Message(E));
end Újra;
```

Végül jöjjön egy példa arra, hogyan lehet kivételeket átadni sablon paraméternek.[11] (11.)

6.7. Példa

```
generic
  Kivétel: Exception_Id;
procedure A;
```

[10]Sajnos ezt a példát az általam használt fordító – gnat 3.09 – elrontja.

[11]Megjegyezzük, hogy Ada83-ban ennek a lehetősége nem állt fenn. Egy alprogram sablonnak például nem lehetett kivételt átadni paraméterként: helyette be kellett ágyazni egy sablon csomagba, amiben rajta kívül a kiváltandó kivétel volt deklarálva.

```
procedure A is
begin
  Raise_Exception(Kivétel);
end A;

procedure CE_A is new A(Constraint_Error'Identity);
```

Ezzel a technikával igen elegánsan készíthetjük el azt a függvényt, ami egy vektor maximumát adja vissza. (Lásd 257. o.)

6.3. Az Ada kivételkezelésének hiányosságai

Vannak más programozási nyelvek is, amelyek támogatják a kivételkezelést, például a Java, az Eiffel, vagy az oly régi PL/1. Érdemes ezekkel összehasonlítani az Adát, mert így könnyebben megérthetjük az Ada kivételkezelési mechanizmusának hiányosságait is.

Az egyik legnagyobb probléma az, hogy elég nehéz a kivétel tényén és fajtáján kívül egyéb információkat átadni a kivételt lekezelni szándékozó programrésznek. Az Ada95-ben már ugyan használhatjuk a kivétel ismertetőjét (`Exception_Message`) szöveges formában kódolt információ átvitelére, de még ez is meglehetősen kényelmetlen. Emiatt sokszor globális változók használatára fanyalodik a programozó a kivétel fellépésének körülményeit jellemző változók eltárolásához. A Java-ban a kivételek objektumok, amik rendelkezhetnek adattagokkal, amelyekben az ilyen jellegű információk a kivétel példánnyal együtt továbbadhatók a hívónak.

További előnye a Java kivétel példányok objektum voltának az, hogy a különböző kivételek (öröklődési) hierarchiába szervezhetőek. Ez sok esetben megkönnyíti a kivételkezelő ágak megírását, jobban támogatja a kód újrafelhasználást, és a kód redundanciáját csökkenti.

Nagyon veszélyes az, hogy a kivételek terjedésének határa nem specifikált a kódban. Előfordulhat, hogy egy programrész egy kivételt nem kezel le, és az többszörösen továbbterjedve végül olyan programrészbe is eljut, ahol nem is gondolnánk, ahová a programozó nem is várta, és ahol már nem is kezelhető le értelmes módon. Ez az effektus Java-ban nem következhet be, hiszen ott egy műveletnek specifikálnia kell, milyen kivételeket vált ki, és a műveletet hívó másik programegységnek azokat vagy le kell kezelnie, vagy neki is specifikálnia kell az általa kiváltott kivételek között.

Ellenkező előjelű problémát okoz a `when others => null;` struktúra fegyelmezetlen használata. Mivel ez minden fellépő kivételt lekezel (helyesebben mondva elfed), esetleg olyanokat is, amelyeket „másnak szántak”, elképzelhető, hogy egy kivétel nem jut el a „címzetthez”.

Utolsó gondolatként említsük meg a Java kivételkezelésének `finalize` szerkezetét, aminek szintén nincs megfelelője az Adában. Java-ban egy kivételkezelt utasítássorozat nem csak a védett utasítássorozatból és a kivételkezelő részből áll, hanem kiegészülhet egy harmadik blokkal, a `finalize` résszel. Az ebben feltüntetett utasítások mind normális, mind abnormális (kivétel fellépésével és esetleg lekezelésével járó) lefutás esetén végrehajtódnak.

6.4. Utalás a következő fejezetekre

Ebben a szakaszban egy kis ízelítőt nyújtunk abból, hogy milyen további ismereteket kínál a könyv hátralevő része a kivételekről.

A taszkokról szóló fejezetből (15.) megtudhatjuk, mikor váltódik ki a `Tasking_Error` előre definiált kivétel. Az is kiderül majd, hogy a taszkok határain túlra nem terjednek a kivételek. Amennyiben egy taszk a törzsében fellépő kivételt nem kezeli le, a taszk csendesen eltűnik: egy taszk abnormális befejezése nem feltétlenül jelenti azt, hogy hiba történt. Az is kiderül majd, hogy ha viszont a taszk aktiválása közben lép fel hiba, a taszkot aktiváló programrészben `Tasking_Error` váltódik ki.

A rendszerközeli programozást támogató nyelvi eszközök bemutatása (17) kitér a `Suppress` pragmára. Segítségével a programban az előre definiált futási idejű ellenőrzéseket ki lehet kapcsolni, el lehet nyomni. Ezzel a programozó növelheti a programja hatékonyságát.

6.4.1. Az előre definiált könyvtárak kivételei

Végezetül felsoroljuk az előre definiált könyvtárakban definiált kivételeket. Ezek közül jónéhánnyal a referencia részben találkozhatunk.

`Ada.Tags.Tag_Error`

`Ada.Calendar.Time_Error`

`Ada.Strings`: `Length_`, `Pattern_`, `Index_` és `Translation_Error`

`Ada.Numerics.Argument_Error`

`Ada.IO_Exceptions`: `Status_Error`, `Mode_Error`, `Name_Error`,
`Use_Error`, `Device_Error`, `End_Error`, `Data_Error` és `Layout_Error`

`Ada.Sequential_IO`: `Status_Error`, `Mode_Error`, `Name_Error`,
`Use_Error`, `Device_Error`, `End_Error` és `Data_Error`
(az `Ada.IO_Exceptions`-beliek átnevezése)

`Ada.Direct_IO`: `Status_Error`, `Mode_Error`, `Name_Error`, `Use_Error`,
`Device_Error`, `End_Error` és `Data_Error`
(az `Ada.IO_Exceptions`-beliek átnevezése)

`Ada.Storage_IO.Data_Error`

`Ada.Text_IO`: az `Ada.IO_Exceptions`-beliek átnevezése

`Ada.Streams.Stream_IO`: `Status_Error`, `Mode_Error`, `Name_Error`, `Use_Error`, `Device_Error`, `End_Error` és `Data_Error` (az `Ada.IO_Exceptions`-beliek átnevezése)

`Interfaces.C.Terminator_Error`

`Interfaces.C.Strings`: `Dereference_Error` és `Update_Error`

`Interfaces.C.Pointers.Pointer_Error`

`Interfaces.COBOL.Conversion_Error`

`System.RPC.Communication_Error`

`Ada.Text_IO.Editing.Picture_Error`

6.5. Feladatok

6.1. Írjon olyan alprogramot, amely kiszámolja egy szövegfájlban található számok összegét! A fájl minden sora egy `Integer` típusú értéket kell, hogy tartalmazzon. Azokat a sorokat, amelyek nem felelnek meg ennek a kikötésnek, figyelmen kívül kell hagyni. (Használja a `Text_IO.Get_Line` műveletet, az `Integer'Value` attribútumot, valamint a `Constraint_Error` kivételt!)

6.2. Készítsen el egy olyan alprogramot, amely két, egészekből álló, monoton növekvően rendezett sorozatot összefésül, a rendezést megtartva! Az alprogram vegye észre, és jelezze egy kivétel kiváltásával, ha valamelyik bemenő sorozat nem monoton növekvő. Módosítsa a kapott programot úgy, hogy a kiváltandó kivételt paraméterként át lehessen adni!

6.3. Az `a+b*c+d+e` jellegű (csak összeadást és szorzást – zárójelezést például nem – tartalmazó) kifejezésekhez készítsen kiértékelő függvényt! A kifejezést szintaktikai egységek sorozataként (tömbjeként) definiálja, ahol a szintaktikai egység valamelyik műveleti jel, vagy egy egész szám lehet. (Használjon variáns rekordot!) Hibás kifejezés esetén kivételt kell a függvénynek kiváltania.

6.4. Az előző feladat megoldását módosítva írjon számológép programot. A kiértékelendő kifejezést nem egy sorozatban kell tárolni, hanem a szabványos bemenetről kell beolvasni. A kiértékelést a szintaktikai egységek beolvasásával párhuzamosan kell végezni. Ha a legutóbb beolvasott szintaktikai egység az aktuális részkifejezést hibássá tenné, (a felhasználót figyelmeztetve) újra be kell azt olvasni. (Írja meg a szintaktikai egységek beolvasását úgy, hogy a felhasználó által begépelt szintaktikusan hibás adatokat figyelmen kívül hagyja!)

7. Fejezet

A csomag

A csomag olyan programegység, amelyben logikailag összetartozó egyedek specifikációjára, implementációjára kerül sor. Gyakran tartalmazza például egy típus deklarációját az elemi típusműveletek deklarációjával együtt. A csomag az *osztály*, az objektum, a *tokbazárás* megvalósítására szolgáló nyelvi elem az Adában, felhasználója elől elrejti belső működésének részleteit.

A programegységek egymásba ágyazhatóak, elhelyezhetőek egy másik programegység deklarációs részében, de a *csomag* önálló fordítási egységként is szerepeltethető, és mint ilyen a program részekre bontásának egyik fő eszköze. Könyvtári csomagnak nevezzük azokat a csomagokat, amelyek önálló fordítási egységek.

Nem csak a programot bonthatjuk fel csomagokra, de a túlságosan nagy csomagokat is tovább bonthatjuk önállóan fordítható részekre. Ennek egyik eszköze az, ha a programkönyvtárat hierarchikusan építjük fel. Egyes csomagok más csomagok titkos vagy nyilvános gyermekei, leszármazottai lehetnek.

7.1. A csomag specifikációja

A csomag két részből áll, a specifikációs részből és a törzsből. Az előbbi írja le az exportált információk jellemzőit, az utóbbi tartalmazza és egyben el is rejti a megvalósítást. A specifikációs rész további két fontos részre bomlik, a látható és az átlátszatlan részre[1]. A látható részben megadott összes információ exportálásra kerül, az itt deklarált egyedek `with`, ill. `use` parancsok használatával általában más programegységekben is használhatóak. Az átlátszatlan részben, ill. a törzsben definiált egyedek csak a csomag deklarációs környezetén belül használhatóak, azaz láthatósági és hatáskörük a csomagon kívülre általában nem terjeszthető ki [2].

[1] A csomag specifikációs részének mindig van átlátszatlan része, legfeljebb üres.

[2] A gyermekcsomagok és a szülő átlátszatlan részének kapcsolatára a későbbiekben még részletesen visszatérünk.

7.1. Példa *(Csomag specifikációs része)*

```
package Verem is
   Tele_Verem , Üres_Verem : exception;
   procedure Belerak(X: in Float);
   function Kivesz return Float;
 private
     --  átlátszatlan rész (esetleg üres)
 end Verem;
```

A csomag specifikációs része deklarációk sorozatát tartalmazza. Mindig a `package` kulcsszóval és a csomag nevével kezdődik. Az átlátszatlan rész kezdetét a `private` kulcsszó jelzi, a csomag végére `end` kerül. A csomag nevét nem kötelező megismételni az `end` után.

7.2. A csomag törzse

A csomag törzse deklarációs részében sorra tartalmazza a specifikációs részben felsorolt programegységek törzseit, tartalmazhat továbbá lokális deklarációkat, valamint tartalmazhat egy inicializáló utasítássorozatot.

Speciálisan előfordulhat az is, hogy a csomagnak csak specifikációs része van, törzse nincs (pl. ha a specifikációs rész csak változókat tartalmaz, és ezek mint közös adatmező használatosak). Kötelezően tartozik azonban a csomaghoz törzs[3], ha van a specifikációs részben olyan deklaráció, amelyik nem teljes és azt kötelező teljessé tenni. Például egy a specifikációs részben deklarált alprogram törzse csak a csomag törzsében lehet, így a csomaghoz biztosan tartozik törzs. Általában is igaz, ha a csomag specifikációja tartalmazza valamely programegység specifikációját, akkor a programegység törzsének a csomag törzsében kell lennie. Az Ada95 általában tiltja törzs készítését könyvtári csomaghoz, ha a törzs nem szükséges[4].

A közvetlenül a törzsben deklarált egyedek abban az értelemben lokálisak, hogy csak a törzsben láthatóak, a csomag külső környezetétől teljes mértékben védettek. Az itt definiált objektumok a csomag törzsének kiértékelésekor jönnek létre, és a csomagot tartalmazó egységből való kilépéskor szűnnek meg. Ezen változók élettartama hosszabb, mint például az alprogramok lokális változóinak élettartama. Közvetlenül könyvtári csomag törzsének deklarációs részében deklarált változók élettartama a program futásának végéig tart, amire gyakran úgy hivatkozunk, hogy ezek a változók „statikusak”.

[3] A törzset az Ada egyes implementációiban megengedett `Import` pragma megadásával helyettesíteni.

[4] Ha a specifikációs részben `Elaborate_Body` pragmát helyezünk el, akkor ebben az esetben is megadható törzs.

Könyvtári szintű csomag esetén a főprogram végrehajtásának megkezdése előtt, beágyazott csomag esetén a tartalmazó deklarációs rész kiértékelésekor kerül sor a törzs kiértékelésére.

Ugyancsak a csomag törzsének kiértékelésekor hajtódik végre a csomag inicializáló része.

A törzsben közvetlenül hivatkozhatunk a törzshöz tartozó specifikációs részben elhelyezett bármely deklarációra.

7.2. Példa *(Csomag törzse)*

```
package body Verem is
    Max : constant := 100;
    S   : array (1 .. Max) of Float;
    Mut : Integer range 0 .. Max;
    procedure Belerak(X: in Float) is
      begin
        if Mut<100 then
          Mut := Mut + 1;
          S(Mut) := X;
        else
          raise Tele_Verem;
        end if;
      end Belerak;
    function Kivesz return Float is
      begin
       if Mut>0 then Mut := Mut - 1;
         return S(Mut + 1);
       else
         raise Üres_Verem;
       end if;
      end Kivesz;
  begin
    Mut := 0;
  end Verem;
```

A `Verem` csomag két alprogramot definiál, amelyek az általa megvalósított (legfeljebb 100 elemű) veremobjektumon végeznek műveleteket. A `Max`, `S`,`Mut` azonosítók lokálisak, a csomagon kívülről nem érhetők el.

Törzs magában nem állhat és minden csomaghoz tartozik egy explicit vagy implicit törzs[5], amely állhat egyetlen `null` utasításból is. Ezáltal minden esetben definiált a csomag törzsében elhelyezkedő utasítássorozat eleje és vége.

[5] Az implicit törzs nem példányosított és nem könyvtári csomag esetén az őt tartalmazó deklarációs egység vége után helyezkedik el. Példány törzse a példányosítás helyén van, egy könyvtári csomag implicit törzsének helyét pedig a feldolgozási függőségek határozzák meg.

7.3. Az átlátszatlan típus

A csomagok egy fontos alkalmazási területe az átlátszatlan típusok, osztályok kialakítása.

Típusok és típuskiterjesztések deklarációja szétválasztható *részleges és teljes nézetre*. A típus vagy kiterjesztett típus részleges nézete (pl. csak a neve) ismert lehet az őt definiáló csomag deklarációs környezetén kívül, de a típus vagy kiterjesztés reprezentációja - a teljes nézet - csak a csomag deklarációs környezetén belül használható. Az átlátszatlan típust vagy típuskiterjesztést a látható részben deklarálhatjuk a `private` kulcsszóval. Ugyanezen egyedek teljes nézetét pedig az átlátszatlan részben kell megadni[6].

7.3. Példa *(Átlátszatlan típus)*

```
package Komplex_Osztály is
  type Komplex is private;
      -- a Komplex típus primitív műveletei
  type Jelölt(d: Diszkrimináns_típus) is
          abstract tagged limited private;
private
  type Komplex is
     -- a Komplex típus reprezentációja
  type Jelölt(d: Diszkrimináns_típus) is
     -- a Jelölt típus reprezentációja
end;
```

A teljes nézet definíciója előtt nem deklarálható a típushoz tartozó változó, nem használható a típus allokátorban, csomag sablon példányosításakor stb. A teljes nézet hatáskörében a teljes nézet határozza meg a típus tulajdonságait, a típust tartalmazó osztályokat, a látható komponensek, belépési pontok, védett eljárások, megengedett attribútumok és előre definiált műveletek halmazát.

Mint a 7.3. példában is megfigyelhetjük, az átlátszatlan típus is paraméterezhető. Pl. a `Text` típus definíciója így írható fel :

```
  type Text(Max:Natural) is private;
```

Ekkor az átlátszatlan részben a típus implementációjának is tartalmaznia kell ugyanezt a paramétert.

[6] A csomag sablonok átlátszatlannak deklarált típusparaméterei úgy tekinthetők, mint amelyeknek a teljes nézetét a példányosítás során definiáljuk, azaz a csomag sablon egy „kifordított csomag". Az egyetlen lényeges különbség, hogy a csomag sablon aktuális paramétere lehet típusosztály, míg a csomagok átlátszatlan típusainak teljes nézete mindig egy-egy konkrét típus. A csomag sablonokat jelentőségüknek megfelelően, részletesen ismertetjük a könyv 11. fejezetében.

(Megjegyzés. A csomag deklarációs környezetén kívül nem hivatkozhatunk az átlátszatlan részben elrejtett információra. A fordítóprogram viszont természetesen felhasználja ezt a részt is a többi egység fordításakor, pl. egy átlátszatlan típusú változó definiálásakor az átlátszatlan részből olvassa el, hogy hány byte-ot foglal el a változó. Így nincs szükség olyan jellegű megszorításra, mint pl. a Modula-2 nyelvben, ahol egy átlátszatlan típus reprezentációja csak annyi helyet foglalhat el, mint a `WORD` típus.)

Azt mondjuk, hogy egy csomag egy *osztályt* valósít meg, ha a csomag egyetlen átlátszatlan típust és ugyanezen típusú objektumokkal paraméterezett alprogramokat tartalmaz. Az átlátszatlan típus reprezentációja gyakran rekord típus. A rekord komponensei írják le az osztályhoz tartozó objektumok közös struktúráját, az alprogramok pedig az üzenetek típusát és a metódusokat. Ilyenkor a csomag neve megállapodás szerint az `_Osztály` utótaggal végződik.

A következő példában megmutatjuk, hogyan építhető fel a `Komplex` osztály, hogyan definiálhatók a komplex számok elemi műveletei, és ezután hogyan használhatja ezeket a programozó.

7.4. Példa *(Osztály)*
```
package Komplex_Osztály is
     type Komplex is private;
     I : constant Komplex;
     function "+"(X,Y:Komplex) return Komplex;
     function "-"(X,Y:Komplex) return Komplex;
     function "*"(X,Y:Komplex) return Komplex;
     function "/"(X,Y:Komplex) return Komplex;
     function Létrehoz(R,I:Float) return Komplex;
     function Valós_Rész(X:Komplex) return Float;
     function Képzetes_Rész(X:Komplex) return Float;
   private
     type Komplex is
       record  RL,IM : Float;
       end record;
     I: constant Komplex :=(0.0,1.0);
   end Komplex_Osztály;
```

A következő példában dinamikus tömböt tartalmazó paraméterezett rekorddal reprezentáljuk az `X` típust:

7.5. Példa *(Paraméterezett átlátszatlan típus)*
```
  package Paraméterezett is
   subtype Index_Tartomány is Integer range 0 .. 100;
   type X(Méret : Index_Tartomány) is private;
       --  az X típus müveletei következnek itt
```

```
private
  type X(Méret: Index_Tartomány) is
     record
       Ay: array (1 .. Méret) of T;
     end record;
end Paraméterezett;
```

Egy 100 hosszúságú vektor típust hoz létre a következő deklaráció:

```
F1 : X(Méret => 100);
```

7.3.1. Késleltetett definíciójú konstansok

A csomag látható részében szerepelhet késleltetett definíciójú konstans is, például:

```
I : constant Komplex;
```

ekkor az átlátszatlan részben meg kell adni ennek az értékét:

```
I : constant Komplex := (0.0,1.0);
```

A késleltetett konstans jól használható átlátszatlan típus konstansának vagy átlátszatlan részben szereplő változó objektumok elérésére, de használata (befagyása) előtt teljessé kell tenni a konstans deklarációját.

Ha egy késleltetett konstans szerepel a látható részben, akkor definíciója ugyanezen csomag átlátszatlan részében, azonos típussal meg kell, hogy jelenjen. Átlátszatlan típushoz tartozó késleltetett konstans esetén először a típus deklarációját kell teljessé tenni, csak utána lehet a konstans értékét definiálni.

Ha a késleltetett konstans deklarációjában a típusa megszorított, akkor a konstans teljes deklarációjának statikusan illeszkednie kell erre. Ha a késleltetett konstans deklarációjában szerepel az `aliased` kulcsszó, akkor a teljes deklarációban is szerepelnie kell.

7.3.2. Az átlátszatlan típusok műveletei

A `Komplex_Osztály` csomag törzse pl. ez lehet :

7.6. Példa *(Átlátszatlan típus műveleteinek megvalósítása)*

```
package body Komplex_Osztály is
 function "+"(X,Y:Komplex) return Komplex is
   begin
    return (X.RL + Y.RL, X.IM + Y.IM);
   end "+";
 function "-"(X,Y:Komplex) return Komplex is
   begin
     return (X.RL - Y.RL, X.IM - Y.IM);
   end "-";
```

```
  function "*"(X,Y:Komplex) return Komplex is
    begin
      return (X.RL * Y.RL - X.IM * Y.IM,
              X.RL * Y.IM + X.IM * Y.RL);
    end "*";
  function "/"(X,Y:Komplex) return Komplex is
     B : Float := Y.RL ** 2 + Y.IM ** 2;
    begin
      return ((X.RL * Y.RL + X.IM * Y.IM) / B,
              (X.IM * Y.RL - X.RL * Y.IM) / B);
    end "/";
  function Létrehoz(R,I:Float) return Komplex is
    begin
      return (R,I);
    end Létrehoz;
  function Valós_Rész(X:Komplex) return Float is
    begin
      return X.RL;
    end Valós_Rész;
  function Képzetes_Rész(X:Komplex) return Float is
    begin
      return X.IM;
    end Képzetes_Rész;
end Komplex_Osztály;
```

Ha komplex számok reprezentációját kicseréljük pl. polárkoordinátás alakra, akkor csak az átlátszatlan részen és a törzsön módosítunk, így a csomagra hivatkozó programegységekben semmit sem kell változtatni.

A `Komplex_Osztály` csomag a következő módon használható fel:

7.7. Példa *(Átlátszatlan típus használata)*

```
with text_Io;use text_Io;
with Komplex_Osztály;use Komplex_Osztály;
procedure Komplex_Próba is
 package fl_Io is new Float_Io(Float);
 use fl_Io;
 Z : constant Komplex := Létrehoz(0.0,0.0);
 C,D:Komplex;
 R,S:Float;
begin
 C := Létrehoz(1.5, -0.6);
 D := C + I;                 -- Komplex +
 R := Valós_Rész(D) + 6.0; -- Float +
```

```
   put(Valós_Rész(C));
   put(Képzetes_Rész(C));
   put(Valós_Rész(D));
   put(Képzetes_Rész(D));
   put(R);
 end Komplex_Próba;
     -- Figyeljük meg, hogy pl. D = I helyes feltétel,
     -- de D > I nem az!
```

Az átlátszatlan típus műveletei az értékadás, az = és a /= relációk, s azok az alprogramok, amiket ugyanabban a specifikációs részben specifikáltunk, ahol az átlátszatlan típus van, s amelyek paraméterlistáján az átlátszatlan típus szerepel. Átlátszatlan típusból való származtatáskor a típusműveletek ugyanúgy átszármaznak az új típusra, mint más típusok esetén.

Az átlátszatlan részben és a csomag törzsében természetesen használható az átlátszatlan típus reprezentációjából adódó minden típusművelet.

Egy típushoz tartozó előre definiált műveletek halmazát meghatározza, hogy a típus melyik típusosztályhoz tartozik. Általában a típus definícióját közvetlenül követi az előre definiált műveletek deklarációja. Az öröklött alprogramok implicit deklarációja szintén közvetlenül a típus definícióját követi. Ezen szabályok alól a következő kivételek érvényesek:

Összetett típusok tulajdonságait részben meghatározzák az összetevők tulajdonságai. A deklaráció pillanatában az összetevők ott látható tulajdonságai érvényesülnek. Ha később, az összetett típus közvetlen hatáskörén belül, valamely komponens típus további tulajdonsága láthatóvá válik, akkor ezen a ponton a megfelelő előre definiált műveletek is deklarálttá válnak. Hasonló szabály érvényes a származtatott típusokra is. Származtatott típusra minden öröklött alprogram a legkorábbi helyen deklarálódik a típus deklarációja után, annak közvetlen hatáskörén belül, ahol a szülő megfelelő deklarációja látható. Ha nincs ilyen hely, akkor nem deklarálódik, de jelölt típusra ekkor is kiválasztódhat.

7.4. A korlátozott átlátszatlan típus

Az átlátszatlan típusokra az értékadás és az =, /= relációk ugyanúgy alkalmazhatóak, mint más típusok esetében. Ez nem mindig előnyös. Gondoljuk meg, mi lesz az összehasonlítás eredménye két verem esetén, vagy mi lesz az értékadás hatása egy mutatókkal megvalósított típus esetén.

A korlátozott átlátszatlan típusú változókra csak a csomagban definiált műveletek alkalmazhatóak. A korlátozott típus[7] absztrakt típusértékek halmaza. Értékeit gyakran a típus egy-egy objektuma reprezentálja, az érték

[7]Pontosabban fogalmazva típusok korlátozott nézetéről beszélhetünk inkább.

és az objektum nem elválaszthatóak, az érték nem másolható le[8]. Ennek megfelelően, korlátozott típushoz tartozó paraméterek átadása mindig cím szerinti, a paramétereknek kezdeti értéke nem lehet.

Típus pontosan akkor korlátozott, ha korlátozottként definiált típus, vagy indexvédett típus korlátozott komponenst tartalmazó összetett típus leszármazottja.

Ha jelölt típus bármely komponense korlátozott, akkor az Ada szabályai megkövetelik, hogy az egésznek is annak kell lennie[9]. Ily módon jelölt típus sohasem veszíti el korlátozottságát. Megjegyezzük, hogy nem jelölt, korlátozott, nem átlátszatlan típus elveszítheti korlátozottságát hatásköre egy részén, kivéve ha átlátszatlan[10].

Például, ha egy tömb típus átlátszatlan és korlátozott komponenstípusának implementációja a tömb típus deklarációjának közvetlen hatáskörén belül láthatóvá válik és az nem korlátozott, akkor a tömb típus is elveszíti korlátozott tulajdonságát ettől a ponttól fogva.

Példaként a korlátozott átlátszatlan típusra tekintsük a következő csomagot.

7.8. Példa *(Korlátozott átlátszatlan típus)*

```
package Verem_Osztály  is
 Tele, Üres : Exception;
 type Verem is limited private;
 procedure Üres_Verem (V: out Verem);
 procedure Kivesz (V: in out Verem; I: out Integer);
 procedure Belerak(V: in out Verem; I: in Integer);
private
  Meret : constant := 100;
  subtype Kis_Számok is Integer range 1..Meret ;
  type Tömb is array (Kis_Számok) of Integer;
  type Verem is
    record
      Elemek : Tömb;
      Mut : integer range 1..Meret+1;
    end record;
end Verem_Osztály;

package body Verem_Osztály is
 procedure Üres_Verem (V: out Verem) is ...
```

[8]Átlátszatlan típusok esetében mindez csak akkor igaz, ha átlátszatlan típusnál a részleges és a teljes nézet is korlátozott.

[9]Ellenkező esetben, ha a komponens teljessé válása során a teljes nézet már nem korlátozott, akkor az egész típus is elveszítheti korlátozott tulajdonságát.

[10]Ebben az esetben is lehet a teljes nézet nem korlátozott és ily módon a típus késleltetett konstansa is kaphat értéket teljes definíciójában.

```
procedure Kivesz(V: in out Verem; I: out Integer) is ...
procedure Belerak(V: in out Verem; I: in Integer) is ...
   ...
begin
  ...
end Verem_Osztály;
```

7.5. Átlátszatlan kiterjesztések

Jelölt típusokat új rejtett komponensekkel bővíthetünk ahhoz hasonlóan, ahogy átlátszatlan típust hozunk létre.

7.9. Példa *(Átlátszatlan típuskiterjesztés)*

```
package Kiterjesztés is
    type Kiterjesztett(d: Diszkriminháns_típus) is
        abstract new Ős_Típus with private;
private
    type  Kiterjesztett(d: Diszkriminháns_típus) is
        -- a   Kiterjesztett típus reprezentációja
end;
```

Az átlátszatlan kiterjesztésekre az átlátszatlan típusokra és a típuskiterjesztésre vonatkozó szabályok együttesen érvényesek.

Ennek megfelelően az átlátszatlan kiterjesztés is örökli az ős komponenseit (általában beleértve a diszkriminánsokat is), az ős primitív eljárásait, ugyanúgy, ahogy a rekord-kiterjesztések örökölnek.

Az ős típus alapján az átlátszatlan kiterjesztés teljes nézete (esetleg több lépésben) levezethető[11].

Átlátszatlan kiterjesztésnél minden öröklött alprogram közvetlenül az átlátszatlan kiterjesztés deklarációja után deklarálódik, ha az ős megfelelő deklarációja látható ezen a helyen. Egyébként az öröklött alprogram nem deklarálódik az átlátszatlan kiterjesztésre.

7.6. Illeszkedési szabályok

Átlátszatlan típus implementációjának illeszkednie kell a típus részleges nézetére. Az illeszkedés szabályait az alábbi táblázatban adjuk meg.

A 2. illeszkedési szabály szigorúbb a harmadiknál, ennek szükségességére az érdeklődő olvasó talál egy példát [2]-ben.

[11] Még akkor is levezethető a teljes nézet, ha csomag sablon példányosítás során keletkeztek egyes közbenső típusok.

	részleges nézet	teljes nézet
1	nem korlátozott	nem korlátozott
2	jelölt,korlátozott	korlátozott
3	nem jelölt, korlátozott	lehet nem korlátozott
4	jelölt	jelölt
5	nem jelölt	jelölt vagy nem jelölt

7.1. táblázat: Részleges és teljes nézet illeszkedése

Ha a részleges nézet nem jelölt és a teljes nézet jelölt, akkor a részleges nézet közvetlen hatáskörében (pl. gyermekcsomagban) annak leszármazott típusa nem megengedett. Csak teljes nézet alapján lehet leszármazott típust deklarálni. Egy nem jelölt, átlátszatlan típus (nem közvetlen hatáskörben deklarált) leszármazottja nem jelölt, még akkor sem, ha a típus teljes nézete jelölt és típuskonverzióval a teljes nézet elemei elérhetőek.

Ha a részleges nézetben szerepel egy diszkrimináns, akkor ennek meg kell jelennie a teljes nézetben is[12].

A részleges nézet őse lehet megszorítás nélküli. A teljes nézet szülő típusa pontosan akkor megszorított, ha az ős altípus megszorított. Ha az ős altípus megszorított, akkor ennek statikusan megfelelő megszorítás kell, hogy megjelenjen a teljes nézetben a szülő típusában. Ha a teljes nézet szülő típusának megszorítása függ a teljes nézet diszkriminánsai értékétől, akkor az ős altípus nem lehet megszorított mert ilyenkor az egyezés nem statikus, hanem értékfüggő.

`S` legyen egy nem jelölt, átlátszatlan típus altípusa, amelynek teljes nézete jelölt. Ekkor az `S'Class` attribútum egy olyan osztályszintű altípust definiál, amelyik a teljes nézetnek felel meg. Ezért az attribútum használata csak az átlátszatlan rész kezdetétől megengedett.

7.7. Példaprogramok

7.7.1. Véletlenszámok előállítása

A csomag a nyelv legalkalmasabb eszköze absztrakt fogalmak megvalósítására. Nem a típusmegvalósítás az egyetlen terület, ahol ilyen célra alkalmazhatjuk a csomagot. Ebben a pontban egy másfajta absztrakt fogalom implementálására mutatunk példát.

[12] A csomag sablon analógia teljessége érdekében az Ada95 megengedi, hogy a diszkrimináns nélkül megadott átlátszatlan típusnak is lehessen diszkriminánssal definiált implementációja, ha a diszkriminánsnak van default értéke. Átlátszatlan, nem jelölt, diszkrimináns nélkül megadott típusnak is lehet jelölt, diszkriminánst is tartalmazó implementációja, de csak akkor, ha a teljes nézet megszorított, mert jelölt típusban nem lehetnek megadva kezdeti értékek.

```
package Rng is
   function Random return Float;
   Seed : Integer;
end Rng;

with Calendar; use Calendar;
package body Rng is
     Modulus : Integer := 65536;
     Mult : Integer := 13849;
     Add_On : Integer := 56963;
   function Random return Float is
   begin
     Seed := (Seed * Mult + Add_On) mod Modulus;
     return Float(Seed) / Float(Modulus);
   end Random;
begin
 Seed := Integer(Seconds(Clock)) mod Modulus;
end Rng;
```

7.7.2. Halmaz megvalósítása

A `Súlyozott_Halmaz` csomag egy olyan halmazobjektumot valósít meg, amelynek elemeit az `Azonosítók` diszkrét típus értékeivel, mint kulcsokkal azonosítjuk, s ezekhez a kulcsokhoz `Duration` típusú értékekkel megadott időtartamok tartoznak, mint súlyok. `Legkisebb` alprogramot függvényként megvalósítani nem volna szerencsés, ennek ugyanis mellékhatása van: azon kívül, hogy a legkisebb elem indexét megadja, ezt az elemet ki is veszi a halmazból.

7.10. Példa *(Súlyozott halmaz objektum)*

```
package Súlyozott_Halmaz is
   type Azonosítók is range 1 .. 128;
   subtype Súly is Duration;
   procedure Belerak(Elem: in Azonosítók; T: in Duration);
   procedure Legkisebb(Elem: out Azonosítók);
   function Üres_e return Boolean;
end Súlyozott_Halmaz;

package body Súlyozott_Halmaz is
   Eleme_e : array (Azonosítók) of Boolean
             := (Azonosítók => False);
   Súlyok : array (Azonosítók) of Súly;

   procedure Belerak(Elem: in Azonosítók; T: in Súly) is
```

```
  begin
   Eleme_e(Elem) := True; Súlyok(Elem) := T;
  end Belerak;

   procedure Legkisebb(Elem: out Azonosítók) is
   -- Feltételezzük, hogy a halmaz nem üres.
   T : Súly := Duration'Last;
   Min : Azonosítók;
   begin
     for I in Azonosítók loop
      if Eleme_e(I) and then Súlyok(I) <= T then
        Min := I; T := Súlyok(I);
      end if;
     end loop;
     Eleme_e(Min) := False;
     Elem := Min;
   end Legkisebb;

  function Üres_e return Boolean is
     begin
        for I in Azonosítók loop
          if Eleme_e(I) then return False; end if;
        end loop;
        return True;
  end Üres_e;
end Súlyozott_Halmaz;
```

Ez a csomag egyetlen halmazobjektumot valósít meg. Ennél általánosabb megoldás az, amit a következő példában mutatunk a Kupac típusra.

7.7.3. A Kupac osztály megvalósítása

A Kupac osztály egyetlen komponens típust tartalmazó iterált osztály. A kupac alaptípusa csak olyan típus lehet, amelyen értelmezve van egy rendezési reláció. A típushoz öt műveletet definiálunk, mindegyiket függvény formájában:

Létrehoz	$\rightarrow$ Kupac
Belerak	: Kupac $\times$ Elemtípus $\rightarrow$ Kupac
Legnagyobb	: Kupac $\rightarrow$ Elemtípus
Kivesz	: Kupac $\rightarrow$ Kupac
Üres_e	: Kupac $\rightarrow$ Boolean

Az osztály sajátossága, hogy a `Legnagyobb` és `Kivesz` művelet mindig a kupacot reprezentáló sorozat legnagyobb elemét választja le. Az egyes műveletek jelentését az alábbi egyenletekkel határozzuk meg:

$$
\begin{array}{ll}
\text{Üres_e(Létrehoz)} & = \text{Igaz} \\
\text{Üres_e(Belerak(h,x))} & = \text{Hamis} \\
\text{Legnagyobb(Létrehoz)} & = \text{Hiba} \\
\text{Kivesz(Létrehoz)} & = \text{Hiba} \\
\text{Belerak(Kivesz(h),Legnagyobb(h))} & = \text{h} \\
\text{Kivesz(Belerak(h,x))} = & \\
\quad = \begin{cases} \text{h, ha Üres_e(h)} \vee \text{Legnagyobb(h)} \leq \text{x} \\ \text{Belerak(Kivesz(h), x), különben} \end{cases} & . \\
\text{Legnagyobb(Belerak(h,x))} = & \\
\quad = \begin{cases} x\text{, ha Üres_e(h)} \vee \text{Legnagyobb(h)} \leq \text{x} \\ \text{Legnagyobb(h), különben} \end{cases} & .
\end{array}
$$

Egy olyan reprezentációt választunk, amelyben a legnagyobb elem könnyen elérhető, de egy elem behelyezése vagy törlése is kevés műveletet igényel. A kupacot egy `h(1..SIZE)` vektorral és egy `n` egésszel ábrázoljuk. `n` a sorozat hossza és `n = 0` jelenti az üres kupacot, a kupac elemeinek megfelelő sorozatot pedig a `h(1)`, ..., `h(n)` elemekkel fejezzük ki. A reprezentációhoz hozzárendelt invariáns: $\forall 1 \leq i \leq \frac{n}{2} : h_i \geq max(h_{2i}, h_{2i+1})$.

Ez a vektorral való ábrázolás megfelel a kupac egy kiegyensúlyozott rendezett bináris fával való ábrázolásának. Az invariáns azt jelenti, hogy minden részfában a gyökér a legnagyobb elem. Így a sorozat legnagyobb elemét mindig `h(1)` ábrázolja, annak kivételekor pedig `h(2)` vagy `h(3)`, ezek helyére pedig a „belőlük lelógó" elemek egyike „csúszik előre", és így tovább. A Belerak, Kivesz függvényeket hatékonysági okokból eljárás formájában kódoljuk.

7.11. Példa *(Kupac osztály)*

```
package Kupac_Osztály is

  subtype Kis_Szám is Natural range 1..255;
  type Kupac(Méret : Kis_Szám := 128) is limited private;
  Üres_Kupac : exception;
  Tele_Kupac : exception;
  procedure Belerak(H : in out Kupac; X : in Integer);
  function  Legnagyobb(H : Kupac) return Integer;
  procedure Kivesz(H : in out Kupac);
  function  Üres_e(H : Kupac) return Boolean;

private
   type Vektor is array (Integer range <>) of Integer;
   type Kupac(Méret : Kis_Szám := 128) is
      record
          N : Natural := 0;
          H : Vektor(1 .. Méret);
      end record;
end Kupac_Osztály;
```

```
package body Kupac_Osztály is
   procedure Belerak(H : in out Kupac; X : in Integer) is
      I, J : Natural; Csere : Boolean;
   begin
    if H.N = H.Méret then raise Tele_Kupac; end if;
    H.N := H.N + 1; J := H.N; Csere := True;
    while Csere and then J >= 2 loop
        I := J; J := J / 2;
        if X > H.H(J)
         then H.H(I) := H.H(J);
         else Csere := False;
        end if;
    end loop;
    if Csere then H.H(J) := X; else H.H(I) := X; end if;
   end Belerak;

  function Legnagyobb(H : Kupac) return Integer is
    begin
     if H.N = 0 then raise Üres_Kupac; end if;
        return H.H(1);
    end Legnagyobb;

  procedure Kivesz(H : in out Kupac) is
      I, J : Natural; Csere : Boolean; Tmp  : Integer;
   begin
     if H.N = 0 then raise Üres_Kupac; end if;
     Tmp := H.H(H.N); H.N := H.N - 1;
     J := 1; Csere := True;
     while Csere and then 2*J <= H.N loop
       I := J; J := 2 * J;
       if J < H.N and then H.H(J+1) > H.H(J)
          then J := J + 1;
       end if;
       if Tmp < H.H(J)
         then H.H(I) := H.H(J);
         else Csere := False;
       end if;
      end loop;
      if Csere
        then H.H(J) := Tmp;
        else H.H(I) := Tmp;
      end if;
    end Kivesz;
```

```
    function  Üres_e(H : Kupac) return Boolean is
       begin
         return (H.N = 0);
       end Üres_e;

end Kupac_Osztály;
```

7.8. Ellenőrzött típusok

Összetett objektumok esetén egyetlen értékadással nem mindig lehet megfelelően definiálni annak kezdeti értékét. Az Ada 95 lehetőséget biztosít arra, hogy az objektumok létrehozásakor, felszámolásakor szükséges műveleteket egy osztály minden objektumára egységesen definiáljuk. Ellenőrzött típusú objektumok esetén a típus műveletei között megtaláljuk az `Initialize` , a `Finalize`, és az `Adjust` műveleteket, amelyek a típushoz tartozó objektumok, lezárásakor, létrehozásakor ill. értékadás során hajtódnak végre.

Ellenőrzött típus a `Controlled` vagy a `Limited_Controlled` típus leszármazottja, amely típusok az `Ada.Finalization` csomagban definiáltak. Műveletei: `Finalize`, `Initialize` és `Adjust`[13], a típus absztrakt, jelölt és átlátszatlan, ill. korlátozott is. A műveletek nem absztraktak ugyan, de közvetlenül ezen típushoz tartozó objektumok sem létezhetnek. A műveletek előre definiált implementációja nem változtatja meg az objektum állapotát, az `=` művelet mindig igazat ad.

Minden objektumot inicializálunk[14] miután pl. objektum deklarációval vagy allokátorral létrehoztuk. Explicit inicializálás esetén a deklaráció vagy allokátor feldolgozásakor a kezdőérték másolása és illesztése következik be, különben pedig ellenőrzött típusú objektum vagy objektumkomponens esetén a megfelelő `Initialize` eljárás hajtódik végre.

Minden objektumot lezárunk mielőtt felszámoljuk. Objektumokat felszámolunk pl. alprogramtörzs elhagyásakor vagy `Unchecked_Deallocation` használata esetén.

Értékadás történik az értékadó utasításban, explicit inicializálásban, paraméterátadáskor stb. Értékadás során a célobjektum lezárásra kerül mielőtt az új érték bemásolására és illesztésére sor kerül.

A nyelv ezen műveleteket minden típusra előre definiálja, de az ellenőrzött típus további lehetőséget biztosít a programozó számára.

Ellenőrzött típus objektumára az előre definiált inicializálás után azonnal meghívásra kerül a `Initialize` eljárás, a `Finalize` eljárás pedig közvetlenül

[13] csak a `Controlled` esetén
[14] default módon vagy explicit

az objektum komponensei előre definiált lezázása előtt kerül meghívásra, továbbá az értékadás utolsó lépéseként végrehajtásra kerül az `Adjust` eljárás.

Az `Initialize` és `Adjust` hívások az objektum szerkezete szerint alulról felfelé, a `Finalize` hívások felülről lefelé hajtódnak végre. Ez a rendezés a komponensek jól definiált állapotához kell. Dinamikus típusú diszkriminánstól függő komponenseket később inicializál a rendszer, mint azoktól nem függőket. Ilyen esetben a diszkriminánst tehát előbb inicializálja, mint amikor hivatkozás történik rá.

Öröklődés esetén a `Finalize` általában úgy működik , hogy először a szülő `Finalize` műveletét hívja meg [15].

Értékadás elvben egy átmeneti névtelen objektum létrehozásával megy végbe. A kifejezés kiértékelése és a szükséges konverzió elvégzése után a névtelen objektumba másolódik és illesztésre kerül az érték. Ezek után a cél lezárásra kerül, majd a névtelen objektum értékét kapja meg a célobjektum. Végül a névtelen objektum is lezárásra kerül.

Ha az objektum önmagát kapja értékül, akkor az implementáció eltekinthet az `Adjust` és a `Finalize` - esetleg mellékhatásokat is kiváltó - végrehajtásától.

Lezárási műveletekre kerül sor pl. ún. mester egyedek[16] komplетté válása és terminálása közötti időben. A mester egyed komplетté válása után a rendszer bevárja a számára lokális[17] objektumok lezárását és a lokális taszkok terminálását. A lezárások az objektumok létrehozásához képest fordított sorrendben mennek végbe.

Objektum lezárásának elemi típus esetében nincs hatása, ellenőrzött típus esetén a megfelelő `Finalize` eljárás hívódik meg. Összetett típus esetén a komponensek lezárására is sor kerül, ezen belül a sorrend `access` típusú diszkrimináns jelenlététől eltekintve tetszőleges[18].

Megjegyezzük, hogy egy-egy objektum lezárása után nem szabadul fel rögtön a hozzá tartozó tárterület, mert még hivatkozhat rá egy másik, még le nem zárt objektum.

Ha vezérlésátadás során a lezárás közben lép fel kivételes esemény, akkor az biztosan a tartalmazó egység lezárása után és a normál végrehajtás folytatása előtt jelentkezik.

7.9. Gyermekcsomagok

A könyvtári egység önállóan fordított programegység, csomag, sablon csomag, alprogram.

[15] Ezért nem lehet absztrakt a `Controlled` típus előre definiált `Finalize` művelete.

[16] taszk törzs, blokk utasítás, alprogram vagy belépési pont törzs, accept utasítás

[17] Megjegyezzük, hogy dinamikus objektumok keletkezési helye nem a deklarációjuk helyén, hanem a végső őstípus deklarációja szerinti helyen van!

[18] Védett típus esetén a lezárás hatását külön tárgyaljuk a védett típus ismertetésénél.

A könyvtári egység deklarációja és törzse egy-egy könyvtári elem. A programkönyvtár könyvtári egységek és részegységek halmaza. Könyvtári egységeket hierarchikusan szervezhetünk. Könyvtári egységbe logikailag beágyazhatunk más könyvtári egységeket, a gyermekeit. A gyökér könyvtári egység gyermekeivel és unokáival egy alrendszert képez. Minden könyvtári egységnek van szülője a `Standard` kivételével. A gyökér könyvtári egységek szülője a `Standard`. Gyermekegységek a szülőben deklarált egyedek teljes nézetének megfelelő deklarációt látják, így bennük szabadon használhatóak a késleltetett konstansok és az átlátszatlan típusok.

A program részekre bontásáról és felépítéséről részletesen a program szerkezetéről szóló fejezetben írunk.

A gyermekcsomag lehetővé teszi a szülő változtatás nélküli bővítését, megengedve az eredeti csomag átlátszatlan részéhez való hozzáférést, specializációját az eredeti csomag változtatása nélkül. Gyermekcsomagok alkalmazásával lehetőségünk van nagy csomagok részekre bontására.

Az öröklődés nem engedi az átlátszatlan típus teljes nézetében definiált komponensek elérését, a gyermek a szülő deklarációs környezetébe tartozik. Ha pl. egy osztályt új művelettel szeretnénk bővíteni, akkor az osztályt megvalósító csomag kódjának átírása helyett gyermekkönyvtárat hozhatunk létre. Ily módon az eredeti osztályt nem szükséges újrafordítani és önmagában újra tesztelni. A teljes rendszert azonban alaposan tesztelni kell. A gyermek és a gyermeken keresztül esetleg további egységek ugyanis módosítani tudják a szülő – egyébként rejtett – egyed vagy osztályszintű attribútumait is. Az egységek közötti kölcsönhatásokat valamelyest az korlátozza, hogy a gyermek elérheti ugyan az ősök átlátszatlan részeit, de nem a saját bárki által látható részéből.

A gyermekcsomag neve két részből áll, tartalmazza a szülő nevét prefixként, ponttal elválasztva a gyermek saját azonosítójától. Átnevezéssel deklarált könyvtári egység nem szerepelhet szülő egységként, de megadható `with` utasításban.

A *titkos* gyermek csak a leszármazotti fán belül látható, az azon kívüli programrészek elől elrejti az implementációs részleteket, így titkos gyermeken keresztül külső egységek nem férhetnek hozzá rejtett információkhoz sem. Megjegyezzük, hogy ugyanaz az egység különböző tulajdonságú is lehet más-más ősre vonatkoztatva. Pl. egy titkos könyvtári egység nyilvános leszármazottja önmagának és egyben titkos leszármazottja szülőjének. Gyermek csak könyvtári egység lehet, de leszármazott a beágyazott egység is.

A csomag sablonokat másutt tárgyaljuk részletesen, de a gyermekcsomagok és csomag sablonok kapcsolatára itt térünk ki. Könyvtári csomag sablon gyermeke maga is csomag sablon, vagy ugyanezen csomag sablon egy másik gyermekének átnevezése. Ha a könyvtári csomag egy csomag sablon példányosításával jött létre, akkor minden gyermeke is példány vagy könyvtári csomag átnevezése. Példányosított csomag sablon gyermekének már új neve van, a csomag sablon példányának neve a prefix, erre hivat-

kozva lehet a példány gyermekét példányosítani valamely néven. A sablon szerinti viszonyok érvényesek a példányban is. Gyermek átnevezése vagy példányosítása csak a szülő csomag sablon deklarációs környezetén belül lehetséges.

Fordítási egységek szemantikus kapcsolatainak meghatározásakor elegendő a szülőre a `use` utasítást alkalmazni, ha `with` volt a gyerekre. A gyerekre alkalmazott `with` ugyanis a prefixben említett szülőre is vonatkozik. Ha a with parancs említi egy könyvtári egység titkos gyermekét, akkor a hivatkozó fordítási egység csak a könyvtári egység titkos leszármazottjának deklarációja vagy a könyvtári egység leszármazottjának törzse (vagy alegysége) lehet.

A szülő, gyermekei és leszármazottai kapcsolatát a következő példa segítségével mutatjuk be [22]. Legyenek `P, P.C1, P.C2, P.C1.G1` csomagok, ahol `P.C1` és `P.C2` a `P` gyermekei, `P.C1.G1` pedig `P.C1` gyermeke.

P.C1 látható r.	látja	P látható részét
P.C1 átlátszatlan r.	látja	P láth. és átlátszatlan részét
P.C2.G1 láth. r.	látja	P látható és P.C2 látható részét
P.C2.G1 láth. r.	látja	P láth. és átlátszatlan, ill. P.C2 láth. és átlátszatlan részét

7.2. táblázat: Gyermekcsomagok és láthatóság

Ezen túlmenően `with` utasítással más csomagok látható részeit is el lehet érni, pl. szülő is elérheti gyermeke látható részét ily módon. Megjegyezzük, hogy egy gyermekcsomagra kiadott `with` utasítás az összes ősre is implicit vonatkozik.

Ha `P.C2` titkos gyermeke `P`-nek, akkor `P.C1` nem éri el semmilyen módon, `with` utasításban sem szerepelhet `P.C1`-ben.

7.12. Példa *(Gyermekegységek – Ada predefinit könyvtárból[2])*

```
package Rational_Numbers.IO is
  -- a Rational_Numbers csomag nyilvános gyermeke
  procedure Put(R: in Rational);
  procedure Get(R: out Rational);
end Rational_Numbers.IO

private procedure Rational_Numbers.Reduce(R: in out Rational);
 -- a Rational_Numbers csomag titkos gyermekeljárása

with Rational_Numbers.Reduce
  -- hivatkozással a saját titkos gyermekére
package Rational_Numbers is
   ...
end Rational_Numbers;
```

```
with Rational_Numbers.IO;
use Rational_Numbers;
   -- with Rational_Numbers-re nincs szükség!
with Ada.Text_IO;
procedure Próba is
   R : Rational;
begin
  R:=5/3;    -- racionális szám konstrukciója
  Ada.Text_IO.Put("Az eredmény : ");
  IO.Put(R);      -- Rational_Numbers.IO.Put
  Ada.Text_IO.New_Line;
end Próba;

package Rational_IO renames Rational_Numbers.IO;
  -- gyermekcsomag átnevezése
```

7.10. Feladatok

7.1. Készítsen egy olyan csomagot, amelyben az átlátszatlan `Pont`, `Vonal`, `Kör`, `Téglalap` típusokat valósítja meg az `Eltol` és a `Nagyít` műveletekkel.

7.2. Átlátszatlan kiterjesztéssel terjessze ki az előző feladat típusait a térre, azaz hozza létre a `Gömb` stb. típusokat!

7.3. Tegye korlátozottá az előző feladatokban szereplő definiált típusokat!

7.4. Alkalmazzon gyermekcsomagot a 9.4. példában!

8. Fejezet

Hatáskör és láthatóság

Ebben a fejezetben annak a szabályait ismertetjük, hogy az Ada hogyan határozza meg, hogy a programszöveg adott pontján előforduló azonosítóhoz melyik deklaráció tartozik. Az egyes deklarációk hatása, előfordulási helyük szerint más a programszöveg különböző részeinek szemantikájára. Az alábbiakban pontosan meghatározzuk a deklarációk hatáskörét és megadjuk azt is, hogy az egyes deklarációkban bevezetett azonosítókra a programszöveg mely pontjain hivatkozhatunk, definiáljuk azok láthatóságát.

Az Ada95-ben a deklaráció egy egyed egy nézetéhez rendel hozzá azonosítót, azaz nem önmagában az egyedhez köti hozzá az azonosítót, hanem az egyed adott nézetéhez tartozó további tulajdonságokat is jelöli. Ennek megfelelően két különböző azonosító jelölheti ugyanazon egyed két különböző nézetét[1].

8.1. Deklarációs környezet

Az Ada95 a blokkszerkezetű nyelvek leszármazottja, de a nyelv számos olyan nyelvi elemet tartalmaz, amely miatt a logikailag összetartozó programrészek szövegszerűen elválnak egymástól. Ezért az Ada95-ben deklarációk hatáskörének és láthatóságának megadásakor nem támaszkodhatunk egyszerűen az őt tartalmazó blokk fogalmára, hanem szükségünk van programszerkezetek deklarációs környezetének pontos meghatározására. A blokkokhoz hasonlóan a deklarációs környezetek is egymásba ágyazhatóak, de ugyanazon deklarációs környezetbe tartozó szövegrészek gyakran különböző fordítási egységekben jelennek meg. Gyermekcsomagok deklarációja például a szülők

[1] Egyes deklarációk több azonosítót is bevezetnek, pl. ilyen a rekord típus vagy az alprogram deklarációja. Az elsődlegesen bevezetett azonosító a rekord típus, ill. az alprogram neve, a másodlagosan bevezetett azonosítók a komponensekhez tartozó szelektorok, ill. a formális paraméterek azonosítói.

deklarációs környezetéhez tartozik[2]. A deklarációs környezethez tartozó programrészletek sorrendjét is definiálja az Ada[3].

Deklarációs környezet tartozik minden olyan programszerkezethez, amelynek belsejében deklarációkat helyezhetünk el, így a blokk utasításhoz, a ciklus utasításhoz, az `accept` utasításhoz, a kivételkezelőhöz és a legtöbb deklarációhoz[4]. A deklarációs környezetbe beletartozik maga a programszerkezet és általában a beleágyazott programrészlet egésze, továbbá a programszöveg rekurzív definícióval meghatározott - nem feltétlenül összefüggő - része.

Ha a deklarációs környezet tartalmaz egy deklarációt, akkor tartalmazza annak teljes nézetét is, ha az létezik. Ha egy könyvtári egység deklarációs része benne van a deklarációs környezetben, akkor a gyermekegységek deklarációs részei is beletartoznak a deklarációs környezetbe.

Ha egy törzs csonkja beleesik a deklarációs környezetbe, akkor beletartozik az egész alegység is.

Ha egy típus deklarációja eleme a deklarációs környezetnek, akkor a megfelelő rekordreprezentáció előírása is, ha van ilyen.

Egyed vagy nézet deklarációs környezete alatt az egyedet, ill. a nézetét deklaráló deklaráció deklarációs környezetét értjük.

Egy deklarációt közvetlenül tartalmazó deklarációs környezet az, amelyik a legbelső a deklarációt tartalmazó környezetek közül, magához a deklarációhoz rendelt környezetet nem számítva. Deklaráció, illetve az általa deklarált egyed lokális az őt közvetlenül tartalmazó deklarációs környezetre nézve és globális azokra a deklarációs környezetekre nézve, amelyeket az őt közvetlenül tartalmazó deklarációs környezet foglal magába.

8.2. Hatáskör

Deklarációk közvetlen hatásköre a deklaráció helyétől általában az őt közvetlenül tartalmazó deklarációs környezet végéig tart[5]. Megjegyezzük, hogy könyvtári egységek átlátszatlan részében szereplő deklarációk hatásköre nem tartalmazza az egység nyilvános leszármazottainak látható részét, ill. nyil-

[2]Ld. a csomagokról és gyermekcsomagokról szóló fejezetet.

[3]Ha a deklarációs környezetbe egy önállóan fordított csomag deklarációja, törzse és `with` utasítással hivatkozott gyermekei is beletartoznak, akkor a csomag deklarációját a törzs követi, majd végül a gyermekek következnek. Ha nincs a gyermekre vonatkozó `with` utasítás, akkor a gyermek és a csomagtörzs sorrendje tetszőleges. Létezik ily módon a teljes Ada program legfelső szintű deklarációs környezete is, amely a `Standard` csomaghoz rendelt deklarációs környezet.

[4]Nem tartozik deklarációs környezet a felsorolási típus deklarációjához, illetve deklarációk teljes nézetéhez. Felsorolási típus esetén a típushoz tartozó literálok a típussal azonos szinten deklarálódnak.

[5]Másodlagos azonosítók hatásköre az őket bevezető deklarációban szereplő elsődleges azonosító hatáskörével egyezik meg.

vános gyermek deklarációjának hatásköre kiterjed az ős deklarációs környezetének végéig, de a gyermek és minden más könyvtári egység közvetlen hatásköre is csak a tőle szemantikusan függő egységeket foglalja magába. A hatáskör kezdete általában a deklaráció kezdete, de átlapolható[6] alprogramok[7] esetén a hatáskör kezdete az a hely, ahol a profil meghatározottá válik[8].

Deklarációk hatáskörének megadásához először definiálnunk kell, hogy mit értünk egyedek nézetének látható része alatt. Ez a nézet deklarációjának az a szakasza, amelyik kívülről látható deklarációkat tartalmaz, míg a kívülről nem látható deklarációkat tartalmazó részt a nézet átlátszatlan részének nevezzük. Alprogram, példányosított alprogram látható része a profilja. Összetett típus[9] látható részébe beletartozik minden komponensének deklarációja. Mintaegység látható része tartalmazza a formális paramétereket leíró részt, továbbá sablon csomag esetén tartalmazza a csomag specifikációs részének látható részét, ill. eljárás sablon esetén a profilt. Csomag, példányosított csomag sablon, taszk, védett egység látható része az általa tartalmazott deklarációk sorozata a `private` kulcsszóig.

Deklaráció hatásköre mindig tartalmazza annak közvetlen hatáskörét. Olyan deklarációk esetén, amelyek közvetlenül egy külső deklaráció látható részében jelennek meg vagy egy külső deklaráció nyilvános gyermekei, a hatáskör (rekuzíven) kiterjed a külső deklaráció hatáskörének végéig[10].

8.3. Láthatóság

Deklaráció csak hatáskörén belül lehet látható. A szöveg egy adott helyén egy azonosító jelentését a *láthatóság*i és az átlapolási szabályok határozzák meg. Deklarációkat direkt módon láthatónak nevezünk a programszöveg egy adott részén, ha egyetlen azonosítóval vagy műveleti jellel hivatkozhatunk rá. Látható a deklaráció, ha direkt módon látható vagy szelekciós kifejezéssel hivatkozni tudunk rá. Pl. a csomagban definiált alprogramra a csomagon kívül szelekciós formában (`Csomag.Alprogram`) hivatkozhatunk, és ugyanez a helyzet a rekord komponenseivel is.

Egy deklaráció kétféleképpen lehet direkt módon látható, vagy azért, mert a deklaráció közvetlen hatáskörén belül vagyunk vagy azért, mert

[6]Deklarációk átlapolásáról ugyanebben a fejezetben később írunk.

[7]Az egyszerűség kedvéért most csak az alprogramokról beszélünk, de itt és a fejezet hátralevő részében ilyenkor mindig az ún. hívható egyedek mindegyikére gondolunk. Ilyenek pl. az eljárások, a függvények, a műveletek, a belépési pontok.

[8]A profil meghatározása után derül ki, hogy az alprogram melyik eljárásokat lapolja át.

[9]Taszk és védett típus kivételével.

[10]Könyvtári egység hatásköre csak a tőle szemantikusan függő egységekre terjed ki.

deklarációs környezetéhez tartozik[2]. A deklarációs környezethez tartozó programrészletek sorrendjét is definiálja az Ada[3].

Deklarációs környezet tartozik minden olyan programszerkezethez, amelynek belsejében deklarációkat helyezhetünk el, így a blokk utasításhoz, a ciklus utasításhoz, az `accept` utasításhoz, a kivételkezelőhöz és a legtöbb deklarációhoz[4]. A deklarációs környezetbe beletartozik maga a programszerkezet és általában a beleágyazott programrészlet egésze, továbbá a programszöveg rekurzív definícióval meghatározott - nem feltétlenül összefüggő - része.

Ha a deklarációs környezet tartalmaz egy deklarációt, akkor tartalmazza annak teljes nézetét is, ha az létezik. Ha egy könyvtári egység deklarációs része benne van a deklarációs környezetben, akkor a gyermekegységek deklarációs részei is beletartoznak a deklarációs környezetbe.

Ha egy törzs csonkja beleesik a deklarációs környezetbe, akkor beletartozik az egész alegység is.

Ha egy típus deklarációja eleme a deklarációs környezetnek, akkor a megfelelő rekordreprezentáció előírása is, ha van ilyen.

Egyed vagy nézet deklarációs környezete alatt az egyedet, ill. a nézetét deklaráló deklaráció deklarációs környezetét értjük.

Egy deklarációt közvetlenül tartalmazó deklarációs környezet az, amelyik a legbelső a deklarációt tartalmazó környezetek közül, magához a deklarációhoz rendelt környezetet nem számítva. Deklaráció, illetve az általa deklarált egyed lokális az őt közvetlenül tartalmazó deklarációs környezetre nézve és globális azokra a deklarációs környezetekre nézve, amelyeket az őt közvetlenül tartalmazó deklarációs környezet foglal magába.

8.2. Hatáskör

Deklarációk közvetlen hatásköre a deklaráció helyétől általában az őt közvetlenül tartalmazó deklarációs környezet végéig tart[5]. Megjegyezzük, hogy könyvtári egységek átlátszatlan részében szereplő deklarációk hatásköre nem tartalmazza az egység nyilvános leszármazottainak látható részét, ill. nyil-

[2]Ld. a csomagokról és gyermekcsomagokról szóló fejezetet.

[3]Ha a deklarációs környezetbe egy önállóan fordított csomag deklarációja, törzse és `with` utasítással hivatkozott gyermekei is beletartoznak, akkor a csomag deklarációját a törzs követi, majd végül a gyermekek következnek. Ha nincs a gyermekre vonatkozó `with` utasítás, akkor a gyermek és a csomagtörzs sorrendje tetszőleges. Létezik ily módon a teljes Ada program legfelső szintű deklarációs környezete is, amely a `Standard` csomaghoz rendelt deklarációs környezet.

[4]Nem tartozik deklarációs környezet a felsorolási típus deklarációjához, illetve deklarációk teljes nézetéhez. Felsorolási típus esetén a típushoz tartozó literálok a típussal azonos szinten deklarálódnak.

[5]Másodlagos azonosítók hatásköre az őket bevezető deklarációban szereplő elsődleges azonosító hatáskörével egyezik meg.

vános gyermek deklarációjának hatásköre kiterjed az ős deklarációs környezetének végéig, de a gyermek és minden más könyvtári egység közvetlen hatásköre is csak a tőle szemantikusan függő egységeket foglalja magába. A hatáskör kezdete általában a deklaráció kezdete, de átlapolható[6] alprogramok[7] esetén a hatáskör kezdete az a hely, ahol a profil meghatározottá válik[8].

Deklarációk hatáskörének megadásához először definiálnunk kell, hogy mit értünk egyedek nézetének látható része alatt. Ez a nézet deklarációjának az a szakasza, amelyik kivülről látható deklarációkat tartalmaz, míg a kivülről nem látható deklarációkat tartalmazó részt a nézet átlátszatlan részének nevezzük. Alprogram, példányosított alprogram látható része a profilja. Összetett típus[9] látható részébe beletartozik minden komponensének deklarációja. Mintaegység látható része tartalmazza a formális paramétereket leíró részt, továbbá sablon csomag esetén tartalmazza a csomag specifikációs részének látható részét, ill. eljárás sablon esetén a profilt. Csomag, példányosított csomag sablon, taszk, védett egység látható része az általa tartalmazott deklarációk sorozata a `private` kulcsszóig.

Deklaráció hatásköre mindig tartalmazza annak közvetlen hatáskörét. Olyan deklarációk esetén, amelyek közvetlenül egy külső deklaráció látható részében jelennek meg vagy egy külső deklaráció nyilvános gyermekei, a hatáskör (rekuzíven) kiterjed a külső deklaráció hatáskörének végéig[10].

8.3. Láthatóság

Deklaráció csak hatáskörén belül lehet látható. A szöveg egy adott helyén egy azonosító jelentését a *láthatósági* és az átlapolási szabályok határozzák meg. Deklarációkat direkt módon láthatónak nevezünk a programszöveg egy adott részén, ha egyetlen azonosítóval vagy műveleti jellel hivatkozhatunk rá. Látható a deklaráció, ha direkt módon látható vagy szelekciós kifejezéssel hivatkozni tudunk rá. Pl. a csomagban definiált alprogramra a csomagon kívül szelekciós formában (`Csomag.Alprogram`) hivatkozhatunk, és ugyanez a helyzet a rekord komponenseivel is.

Egy deklaráció kétféleképpen lehet direkt módon látható, vagy azért, mert a deklaráció közvetlen hatáskörén belül vagyunk vagy azért, mert

[6]Deklarációk átlapolásáról ugyanebben a fejezetben később írunk.

[7]Az egyszerűség kedvéért most csak az alprogramokról beszélünk, de itt és a fejezet hátralevő részében ilyenkor mindig az ún. hívható egyedek mindegyikére gondolunk. Ilyenek pl. az eljárások, a függvények, a műveletek, a belépési pontok.

[8]A profil meghatározása után derül ki, hogy az alprogram melyik eljárásokat lapolja át.

[9]Taszk és védett típus kivételével.

[10]Könyvtári egység hatásköre csak a tőle szemantikusan függő egységekre terjed ki.

egy use utasítással direkt láthatóságát közvetlen hatáskörén túl is kiterjesztettük.

Egy könyvtári csomag vagy csomag sablon deklarációja direkt módon látható az is szótól kezdve a csomag deklarációs részében, a csomag törzsében, a leszármazottai deklarációs részében és törzsében, valamint minden olyan fordítási egységben, amelyhez tartozó with utasításban megnevezik.

Egy könyvtári alprogram vagy alprogram sablon direkt módon látható a megfelelő alprogramtörzsben és minden olyan fordítási egységben, amelyhez tartozó with utasításban megnevezik.

Egy nem könyvtári egység deklarációja csak akkor látható direkt módon a közvetlen hatáskörének egészén, ha más deklaráció nem takarja el, ill. nem definiálja felül.

Ha adott deklaráció hatáskörén belül az általa bevezetett azonosítót egy újabb deklarációban – egy beágyazott deklarációs környezetben – ismételten bevezetjük, akkor általában a belső deklaráció saját hatásköre végéig eltakarja a külsőt, az direkt módon nem látható.

Az alábbi példában egy érvényes deklaráció hatáskörében, belső blokkban ugyanahhoz az azonosítóhoz egy másik egyedet rendelünk, ezért a kívül deklarált egyedre csak szelekciós jelöléssel hivatkozhatunk.

8.1. Példa *(Direkt láthatóság eltakarása)*

```
procedure P is
   A,B:Boolean;
   procedure Q is
      C:Boolean;
      B:Boolean;
   begin
     B := A;      --  Q.B := P.A
     C := P.B;    --  Q.C := P.B
   end Q;
begin
   A:=B;          --  P.A := P.B
end P;
```

Itt Q -ban a P -beli B-re csak szelektíven lehet hivatkozni, mert B Q-beli deklarációja eltakarja a P-belit.

8.2. Példa *(Hatáskör és láthatóság)*

```
procedure P is
   A, B, C : Integer;
   procedure Q is
      A, C : Integer;
      procedure R is
         A : Integer;
```

```
      begin
        --  közvetlenül látható nevek: P-beli Q és B,
        --        Q-beli R és C, R-beli A,
        --  csak szelekciós hivatkozással
        --        megadható nevek: P.A, P.C, Q.A;
        --  globális nevek: P, P.A, P.B, P.C, Q, Q.A, Q.C;
        --  lokális nevek: R, A;
      end R;
   begin
        -- közvetlenül látható nevek: P-beli Q és B,
        --      Q-beli R, A  és C
        -- csak szelekciós hivatkozással
        --      megadható nevek: P.A, P.C;
        -- globális nevek: P, P.A, P.B, P.C;
        -- lokális nevek: Q, A, C, R;
   end Q;
   procedure S is
      B : Integer;
   begin
     -- közvetlenül látható nevek: P-beli Q, A, C és S,
     --       S-beli B,
     -- csak szelekciós hivatkozással megadható nevek: P.B;
     -- globális nevek: P, P.A, P.B, P.C, Q;
     -- lokális nevek: S, B;
   end S;
 begin
     -- közvetlenül látható nevek: P-beli Q,S,A,B,C
     -- lokális nevek: P, A, B, C, Q, S;
 end P;
```

Két vagy több deklarációt akkor nevezünk átlapoltnak, ha az általuk bevezetett azonosító megegyezik és van a programszövegnek olyan szakasza, amelyiken mindegyik deklaráció direkt módon látható. Az Ada általában nem engedi meg azonosítók átlapolását, csak a felsorolási konstansok nevei és az alprogramok nevei átlapolhatóak. Ilyenkor is egyértelműen eldönthetőnek kell lenni, hogy az adott hivatkozás melyik deklarációra vonatkozik. Az azonosító környezete (pl. a paraméterlista) alapján – operátorok felüldefiniálásának esetét kivéve – csak egyetlen olyan deklaráció lehet, ami minden szempontból megfelel a hivatkozásnak, különben a hivatkozás szintaktikus hibát jelent. Egy típus explicit deklarációval felüldefiniált operátora minden esetben eltakarja az öröklődés során implicit módon deklarált műveletet, az örökölt művelet semmilyen módon nem tehető láthatóvá[11].

[11] Arra természetesen van lehetőség, hogy az örökölt művelethez átnevezéssel még a felüldefiniálást megelőzően új azonosítót rendeljünk hozzá.

8.3.1. A use utasítás

A `use` utasítás a *szelekciós hivatkozás* helyett lehetővé teszi a direkt hivatkozást. `use` utasításban hivatkozhatunk egy csomagra vagy egy típusra. `use` utasítás két helyen fordulhat elő: egy fordítási egység szemantikus környezetét leíró környezet definícióban, illetve utasítássorozat elemeként. Az előbbi esetben a `use` utasítás hatásköre általában a környezeti definíció után kezdődik és a fordítási egységet magában foglaló legbelső deklarációs egység végéig tart, az utóbbi esetben pedig megjelenési helyétől az őt közvetlenül magába foglaló deklarációs környezet végéig tart és azon túl nem terjed ki. Egy csomag átlátszatlan részében elhelyezett `use` hatásköre azonban nem terjed ki a nyilvános leszármazottak látható részére.

A `use` utasításban hivatkozott csomagnak vagy típusnak az előző pontban leírt szabályok szerint láthatónak kell lennie. A hivatkozott csomag látható részében szereplő deklarációk, ill. a hivatkozott altípus primitív operátorai a `use` hatáskörén belül direkt láthatóvá válnak egy kivétellel:

ha egy csomagra alkalmazott `use` által direkt módon láthatóvá tett azonosító egy korábbi deklaráció vagy `use` következtében is direkt módon látható és azzal nem átlapolható.

8.3. Példa (`use` *és a hivatkozási szabályok alkalmazása)*

```
package D is
  T,U,V : BOOLEAN;
end D;

package P is
   package E is
     B,W,V : Integer;
   end E;
   procedure Q is
     T,X : Float;
     use D,E;
   begin
     -- Ezen a ponton
     -- T jelentése Q.T és nem D.T,
     -- U jelentése D.U,
     -- B jelentése E.B,
     -- W jelentése E.W,
     -- X jelentése Q.X,
     -- V illegális, D.V vagy E.V alakban kell
     --                          V-re hivatkozni!
   end Q;
begin
   ...
end P;
```

Az alábbi példában a P eljárásban a T típus /= operátora két helyről is látszik, ezért a hivatkozásnál a csomag nevének megadásával egyértelművé kell tenni a hivatkozást:

8.4. Példa *(Operátor átlapolása)*

```
package A is
 type T is new Integer;
end A;
package B is
  subtype T1 is T;
end B;
with A, B; use A, B;
procedure P(X, Y :in T) is
  begin
   if B."/="(X, Y) then ... -- így helyes
  end P;

  type Acc_String is access String;
  function F(X : Integer) return Acc_String;
  function F return Acc_String;
   -- itt F(3)'Address nem egyértelmű kifejezés.
```

8.4. Az átnevezés

Lehetséges az eddig felsoroltakon kívül még egy másik hivatkozási mód is: átnevezhetjük a látható objektumokat.

Az alábbi átnevezések mind szabályosak.

8.5. Példa *(Átnevezés)*

```
function Float_Plusz(Left, Right : Float)
                     return Float renames "+";
function Next(C : Szín)
                 return Szín renames Szín'Succ;
declare
 procedure V_Belerak(X: in Float)
                    renames Verem.Belerak;
 function V_Kivesz return Float
                    renames Verem.Kivesz;
begin
 V_Belerak(X);
  ...
 X := V_Kivesz;
end;
```

Az átnevezés egyaránt használható a szöveg rövidítésére, a nevek közötti konfliktusok feloldására és a végrehajtási idő csökkentésére.

```
L : Integer renames D.EV;
AI : Float renames A(I);
```

Ezután az `AI := AI + 1;` forma használatával elkerüljük az indexkifejezés többszöri kiszámítását. (Ha `I` értéke a programban megváltozik, akkor az `AI` továbbra is a korábbi változót jelenti.)

Feladatok

8.1. Adja meg a változók jelentését 8.3. példa `Q` eljárásának belsejében, ha a `Q` eljárásban a `use D,E;` sort a `use E;` sorra cseréljük!

8.2. Mi a különbség a 8.4. példában `A.F(3)'Address` és `B.F(3)'Address` között?

8.3. Nevezze át a 8.4. példa `F` eljárásait!

9. Fejezet

A program szerkezete

9.1. Partíciók, fordítási egységek

A program partíciók halmaza, amelyek külön címtérben hajtódnak végre, esetleg külön számítógépen. A partíció könyvtári egységek halmaza. Osztott program partíciói konkurens módon hajtódnak végre.

Programegység a csomag, a taszk, a védett egység, a védett belépési pont, a sablon egység (sablon csomag, sablon eljárás), az explicit deklarált alprogram. Bizonyos programegységek önállóan fordíthatóak, ill. egymásba ágyazhatóak.

A fordítási egységek kétfélék: ún. könyvtári elemek (alprogram, csomag vagy sablon egység specifikációja, sablon egység példányosítása, ezek törzse vagy átnevezése) vagy alegység.

Mivel a fordítási egységek között különféle kapcsolatok lehetnek (egy egység importálhat egy másikból, specifikáció-törzs kapcsolat, törzs-alegység kapcsolat), egy egység fordításakor a fordítóprogramnak szüksége van a hivatkozott egységekben definiált bizonyos információkra. Ezért a lefordított egységek az ún. Ada könyvtárba kerülnek, ahol az egységek tárgykódján kívül megőrzésre kerülnek a szükséges szimboltáblák is. A könyvtár nyilvántartja az egységek közötti kapcsolatokat is. A fordítási egységek közötti szemantikus függőségi viszonyok az egységek lehetséges fordítási sorrendjét is meghatározzák.

Ha egy egység fordításakor a fordítóprogram hibát észlel, akkor az egység nem kerül be a könyvtárba. Ha olyan egységet fordítunk, ami már szerepel a könyvtárban, akkor az új egység a régi helyére kerül, s *érvénytelen*né válik a könyvtárban minden olyan egység, ami az új egységtől függ. Az érvénytelenné vált egységek a könyvtáron belül *újrafordíthatók*k (tehát anélkül, hogy egy fordítással bevinnénk a könyvtárba, azaz a fordítás és az újrafordítás két külön művelet).

Minden fordítási egység előtt `with` utasításban kell felsorolni azokat az egységeket, amelyekből importálni akarunk. A `with` a hivatkozott egység által exportált összes deklarációt láthatóvá teszi[1]. A megnevezett csomagban definiált nevekre ezután *csomag.név* formában hivatkozhatunk. Hogy ezt a hosszadalmas formát elkerüljük, a `with` utasítás után szinte mindig alkalmazunk egy `use` utasítást is, ami közvetlenül láthatóvá teszi a csomag exportált neveit. (Ha ilyen módon több csomagból is importáltuk ugyanazt az azonosítót, az nem hiba, majd az azonosító használatakor kell egyértelművé tenni, hogy melyikre hivatkozunk.) [2]

9.1. Példa *(Teljes Ada program)*

```
with Text_IO, Ada.Numerics.Elementary_Functions;
procedure Másodfokú_Egyenlet is
    A,B,C,D : Float;
    use Text_IO, Ada.Numerics.Elementary_Functions;
    package F_IO is new Float_IO(Float); use F_IO;
begin
    Get(A); Get(b); Get(c);
    D := B ** 2 - 4.0 * A * C;
    if D < 0.0 then
     Put("Képzetes gyökök");
    else
     Put("valós gyökök :  X1 = ");
     Put((-B - Sqrt(D)) / (2.0 * A));
     Put(" X2 = ");
     Put((-B + Sqrt(D)) / (2.0 * A));
    end if;
    New_Line;
 end Másodfokú_Egyenlet;
```

Egy programegység akkor fordítható, ha azoknak az egységeknek a specifikációs része, amelyekre ez hivatkozik, már le vannak fordítva. (Az alegységekre vonatkozó szabályokat ld. később). A könyvtárban vannak az előre definiált Ada egységek moduljai is.

A könyvtári rendszer biztosítja, hogy a függetlenül fordított egységek kapcsolata egymásnak megfelelő legyen (ezeket ellenőrzi).

[1] A láthatóságra és hatáskörre vonatkozó szabályokat részletesen tárgyaljuk a könyv 8. fejezetében, itt ismertetjük a `with` és `use` utasítás pontos jelentését is.

[2] A programkészítés szempontjából hasznos lenne, ha importáláskor valamiképpen el lehetne hagyni a programból az importált modulnak a felesleges alprogramjait. Pl. kellemetlen, hogy ha a programozó egy olyan fajta készletet használ, mint a matematikai függvények modulja, akkor annak minden alprogramja hozzászerkesztődik a programhoz, pedig talán csak az `Sqrt` függvényre lenne szükség.

9.2. Könyvtári elemek

A könyvtári egység az önállóan fordított programegység, csomag, alprogram vagy sablon egység. , , . A könyvtári egység deklarációja és törzse tehát egy-egy könyvtári elem.

A könyvtári egységek nevei különbözőek kell legyenek. Ada főprogram olyan könyvtári eljárás lehet, amelynek nincs paramétere[3].

A programkönyvtár könyvtári egységek és alegységek halmaza.

Elég sok modulból álló programoknál azt láthatjuk, hogy a modulok a feladat egy-egy logikai része köré csoportosulnak. Pl. egy fordítóprogram moduljainak egy csoportja a szemantikus elemzéshez, másik csoportja a kódgeneráláshoz stb. tartozik. Az ilyen csoportoknak általában csak egy-két olyan moduljuk van, amiből másik csoportbeli modul importál, a többi modul ezt az egy-két modult szolgálja ki. Könyvtári egységeket hierarchikusan alrendszerekbe szervezhetünk az Ada95-ben. Könyvtári egységbe logikailag beágyazhatunk más könyvtári egységeket, a gyermekeit, leszármazottait. A gyökér könyvtári egység gyermekeivel és unokáival egy alrendszert képez. Minden könyvtári egységnek van szülője a `Standard` kivételével. A gyökér könyvtári egységek szülője a `Standard`.

Tekintsük a következő programot.

9.2. Példa *(Program részekre bontása)*
```
procedure Próba is

 type Szín is (Piros,Kék,Zöld,Sárga);

 package Halmazok is
   type Színhalmaz is private;
   procedure Belerak(C: in Szín; CS: in out Színhalmaz);
      -- ....
   private
     type Színhalmaz is array (Szín) of Boolean;
 end Halmazok;

 package body Halmazok is
   procedure Belerak(C: in Szín; CS: in out Színhalmaz) is
      -- ...
   end Belerak;
      -- ...
 end Halmazok;
```

[3]Az Ada megengedi, hogy több önálló fordítási egységet pl. egyetlen file-ban összefogjunk. A nyelv egyes implementációi azonban megszorítást tehetnek olyan fordításokra, amelyben több fordítási egység is van.

```
use Halmazok;
 A,B: Színhalmaz;
begin
 --
 Belerak(Piros,A);
  -- ......
end Próba;
```

Ezt a programot a következőképpen választhatjuk szét önálló progra-megységekre:

```
package Színek is
 type Szín is (Piros,Kék,Zöld,Sárga);
end Színek;

with Színek; use Színek;
package Színhalmaz_Osztály is
  type Színhalmaz is private;
   procedure Belerak(C: in Szín; CS: in out Színhalmaz);
      -- ....
   private
     type Színhalmaz is array (Szín) of Boolean;
 end  Színhalmaz_Osztály;

package body Színhalmaz_Osztály is
   procedure Belerak(C: in Szín; CS: in out Színhalmaz) is
      -- ...
   end Belerak;
       -- ...
 end  Színhalmaz_Osztály;

 with Színek, Színhalmaz_Osztály;
 use Színek, Színhalmaz_Osztály;
 procedure Próba is
 A,B: Színhalmaz;
 begin
  --
  Belerak(Piros,A);
   -- ......
 end Próba;
```

A gyermekcsomagokról részletesen írunk a csomagokról szóló fejezetben és arra is ott mutatunk példát, hogy hogyan lehet gyermekegységek segítségével tovább részekre bontani a programot.

Egy könyvtári alprogram nem definiálhat felül egy primitív alprogramot és önállóan fordított függvény neve nem lehet műveleti jel [4]. Könyvtári alprogram sohasem primitív, ezért jelölt típus esetén nem lehet dinamikusan kiválasztani gyermek alprogramot.

9.3. Alegységek

Egy csomagban definiált alprogram törzsét kiemelhetjük a csomagból, és önálló fordítási egységet, ún. alegységet képezhetünk. A csomagban az alegység törzsének helyén egy csonk marad. Alegységek csak törzsek önálló fordítását támogatják, a szülőben megvan a csonk, amely az alegység létére és helyére utal, a szülő törzsében szereplő deklarációk is láthatóvá tehetőek az alegységben. Alegységként kiemelhető: csomag, alprogram, taszk, védett egység. A csonknak közvetlenül a törzs deklarációs részében kell lennie.

Például a Verem csomag törzsét[5] a következőképpen is írhatjuk:

9.3. Példa *(Alprogram csonk)*

```
package body Verem is

    Max : constant := 100;
    S   : array (1 .. Max) of Float;
    Mut : Integer range 0 .. Max;
    procedure Belerak(X: in Float) is separate;
    function Kivesz return Float is separate;

  begin
    Mut := 0;
  end Verem;
```

A két alegység ekkor külön fordítható, ezek formája a következő:

```
 separate(Verem)
  procedure Belerak(X: in Float) is
      begin
        if Mut<100 then
          Mut := Mut + 1;
          S(Mut) := X;
        else
          raise Tele_Verem;
        end if;
      end Belerak;
```

[4] Egy leszármazott fordítási egység átnevezhető a szülőjében operátorrá.

[5] Ld. a csomagokról szóló fejezetet.

```
separate(Verem)
 function Kivesz return Float is
     begin
      if Mut>0 then Mut := Mut - 1;
        return S(Mut + 1);
      else
        raise Üres_Verem;
      end if;
     end Kivesz;
```

Egy alegységben a hatáskör és a láthatóság pontosan olyan, mintha az alegység a csonk helyén lenne. Tehát pl. az alegységben minden külön jelölés nélkül hivatkozhatunk olyan típusokra és globális változókra, amelyek a szülőben vannak definiálva.

Egy alegységből szintén el lehet különíteni valamely beágyazott programegység törzsét, azaz alegységnek szintén lehet alegysége. Az alegységek fordítási sorrendjére az a szabály, hogy egy alegység csak akkor fordítható le, ha már lefordítottuk azt az egységet, amelyből az alegység származik. Ha egy törzset vagy alegységet ismételten lefordítunk, akkor újra kell fordítani ezek összes alegységét.

Azt az egységet, amiből az alegységet leválasztottuk, az alegység *szülő*jének hívjuk. A szülők láncolatán visszafelé haladva egy olyan egységhez jutunk, ami már nem alegység, hanem valódi törzs. Ezt a törzset az alegység *ős*ének nevezzük.

Nem egyezhet meg két olyan alegységnek a neve, amelyeknek közös ősük van.

9.4. Szemantikus függőség

Könyvtári egységek szemantikus függőségi viszonyainak pontos szabályai a következők. Könyvtári egység szemantikusan függ a szülője deklarációjától. Alegység szemantikusan függ a szülő törzsétől. Könyvtári egység törzse függ a deklarációtól (ha van olyan, pl. könyvtári alprogramnak nem biztos, hogy van). Fordítási egység szemantikusan függ a `with` utasításban felsorolt könyvtári elemektől. A függés tranzitív.

Fordítási egységek szemantikus kapcsolatainak meghatározásakor elegendő a `use` utasítást alkalmazni a szülőre, ha `with` volt a gyerekre. A `with` utasítás ugyanis a gyermek nevének prefixében említett szülőre is vonatkozik.

Ha a `with` parancs említi egy könyvtári egység titkos gyermekét, akkor a hivatkozó fordítási egység csak a könyvtári egység titkos leszármazottjának

deklarációja vagy a könyvtári egység leszármazottjának törzse vagy alegysége lehet[6].

Fordítási egység `with` parancsában említett könyvtári elem úgy tekinthető, mint egy rendes deklaráció. Beágyazott egység is szerepelhet `use` parancsban.

9.5. Példaprogramok

9.5.1. Prioritási sorok

Ebben a példában több csomag felhasználásával építünk fel egy absztrakt objektumot.

Tekintsünk egy olyan operációs rendszert, amelyben az elvégzendő feladatok – a programban *job*-oknak nevezzük őket – 1 és 10 közötti prioritásokkal vannak ellátva. Az `Prioritássorok` csomagon keresztül fogjuk az elvégzendő feladatokat kezelni.

9.4. Példa *(Prioritássor)*

```
package Job is
  type Job_Azonosító is new Integer;
end Job;

with Job; use Job;
package Prioritássorok is
  Max_Méret : constant := 50;
  type Prioritás is new Integer range 1 .. 10;
  procedure Hozzáad(P: in Prioritás; J: in Job_Azonosító);
       -- A várakozó jobok közé felveszi
       -- a J azonosítójú P prioritású jobot.
  procedure Kivesz(J: out Job_Azonosító);
       -- Kiveszi a legmagasabb prioritású várakozó jobok
       -- közül azt, amelyik a legrégebben várakozik.
  function Tele_e(P : Prioritás) return Boolean;
       -- Egy adott prioritáshoz legfeljebb Max_Méret darab
       -- várakozó tartozhat. Igazat ad vissza, ha a P
       -- prioritáshoz tartozó várakozási sor tele van.
  function Van_e_Job return Boolean;
       -- Igazat ad vissza, ha van várakozó job.
end Prioritássorok;
```

Az egyes prioritásokhoz tartozó várakozási sorok kezelésére elkészítjük a `Sor_Osztály` csomagot.

[6] Külön deklaráció nélküli alprogram ebből a szempontból deklarációnak számít.

9.5. Példa *(Sor_Osztály)*

```
with Job; use Job;
package Sor_Osztály is
  type Sor(Max: Integer) is limited private;
  Sor_Túlcsordulás : exception;
  Sor_Alulcsordulás: exception;
  procedure Belerak(Q: in out Sor; J: in Job_Azonosító);
     -- Hozzáveszi a Q sorhoz a J értéket.
  procedure Kivesz(Q: in out Sor; J: out Job_Azonosító);
     -- Visszaadja és törli a Q sorból a legrégebbi elemet.
  function Tele_e(Q : Sor) return Boolean;
  function Üres_e(Q : Sor) return Boolean;
 private
   type Jobok is array (Integer range <>) of Job_Azonosító;
   type Sor(Max: Integer) is
     record
        X : Jobok(1 .. Max);
        Legrégebbi, Utolsó : Integer := 1;
        Méret : Integer := 0;
     end record;
 end Sor_Osztály;

 package body Sor_Osztály is
    procedure Belerak(Q: in out Sor; J: in Job_Azonosító) is
     begin
       if Tele_e(Q) then raise  Sor_Túlcsordulás; end if;
       Q.X(Q.Utolsó) := J;
       Q.Utolsó := Q.Utolsó mod Q.X'Length + 1;
       Q.Méret := Q.Méret + 1;
     end Belerak;

    procedure Kivesz(Q: in out Sor; J: out Job_Azonosító) is
     begin
       if Üres_e(Q) then raise Sor_Alulcsordulás; end if;
       J := Q.X(Q.Legrégebbi);
       Q.Legrégebbi:= Q.Legrégebbi mod Q.X'Length + 1;
       Q.Méret := Q.Méret - 1;
     end Kivesz;

    function Tele_e(Q : Sor) return Boolean is
     begin
       return Q.Méret = Q.X'Length;
     end Tele_e;
```

```
    function Üres_e(Q : Sor) return Boolean is
     begin
       return Q.Méret = 0;
     end Üres_e;
  end Sor_Osztály;
```

A Prioritássorok csomag törzse most már minden nehézség nélkül megírható a Sor_Osztály csomag felhasználásával.

9.6. Példa *(Prioritássorok törzse)*

```
with Sor_Osztály;
package body Prioritássorok is
  PQ : array (Prioritás) of Sor_Osztály.Sor(Max_Méret);
  procedure Hozzáad(P: in Prioritás; J: in Job_Azonosító) is
    begin
     Sor_Osztály.Belerak(PQ(P), J);
    end Hozzáad;

  procedure Kivesz(J: out Job_Azonosító) is
    begin -- Feltételezi, hogy van várakozó job.
      for I in reverse Prioritás loop
        if not Sor_Osztály.Üres_e(PQ(I)) then
          Sor_Osztály.Kivesz(PQ(I), J);
          return;
        end if;
      end loop;
    end Kivesz;

  function Tele_e(P : Prioritás) return Boolean is
    begin
      return Sor_Osztály.Tele_e(PQ(P));
    end Tele_e;

  function Van_e_Job return Boolean is
    begin
     for I in Prioritás loop
      if not Sor_Osztály.Üres_e(PQ(I))
         then return True;
      end if;
     end loop;
     return False;
 end Van_e_Job;
end Prioritássorok;
```

9.6. Keresztreferencia lista készítése

Ebben a pontban egy Ada83 nyelvű programokról keresztreferencia listát nyomtató programot készítünk. A program névsor szerint felsorolja az Ada programban szereplő azonosítókat, s mindegyikről megadja, hogy mely sorokban fordul elő. A megjegyzéseket, szövegliterálokat és az alapszavakat a program nem veszi figyelembe.

A program két részből áll: egy scanner-ből, ami az input szöveget „megválogatja", kihagyja belőle a feladat szempontjából érdektelen részeket, és egy táblakezelőből, ami nyilvántartja az azonosítókat és a sorszámokat, valamint nyomtatni tudja ezt a táblázatot. Az elemző részt a főmodul foglalja magában, először ezt adjuk meg. A főmodul hivatkozik a táblakezelő package-re, aminek a specifikációs része a következő :

9.7. Példa *(Táblázat_Osztály)*

```
with Text_IO; use Text_IO;
package Táblázat_Osztály is
  type Táblázat is private;
  Tele_Táblázat : exception;

  procedure Üres_Táblázat(T: in out Táblázat);
  procedure Bejegyez(T: in Táblázat; Azonosító: in String;
                     N: in Integer);
     -- Felvesz egy nevet a táblázatba, ha az még nem
     -- szerepelt benne, s hozzáírja a sorszámot.
  procedure Nyomtat(F: in File_Type; T: in Táblázat);
     -- Kinyomtatja a táblázat teljes tartalmát.

private
    type Bejegyzés;
    type B_Mutató is access Bejegyzés;
    type Bejegyzés is record
          Num : Integer := 0;
          Next : B_Mutató;
    end record;
    type S_Mutató is access String;
    type Szó;
    type Táblázat is access Szó;
    type Szó is record
           Kulcs : S_Mutató;
           Els : B_Mutató;
           Bal, Jobb : Táblázat;
    end record;
 end Táblázat_Osztály;
```

A főmodul a táblázatkezelő modulon kívül csak a text input-output modult használja fel.

9.8. Példa *(Keresztreferencia készítés)*

```
with Text_IO, Táblázat_Osztály;
use Text_IO, Táblázat_Osztály;
procedure Xref is

   LNO : Natural := 1; -- az aktuális sor sorszáma
   T : Táblázat;
   F, G : File_Type;
   InFile, OutFile : String(1 .. 64);
   ID : String(1 .. 80);
   CH : Character;
   K : Natural;
   Aposztróf_Után : Boolean := False;

  type Kulcsszó_Típus is access String;
  type Kulcsszavak_Típus is array (Integer range <>)
                        of Kulcsszó_Típus;

  További_Betûk : constant String(1..61) :=
     "           ";
        -- a Latin-1 ábécé ékezetes kis- és nagybetûi

  További_Azonosító_Betû : array (Character) of Boolean
             := (others =>False);
    -- A Feltölt_További_Betük eljárás tölti ki

  Kulcsszavak : constant Kulcsszavak_Típus(1 .. 69) := (
        new String'("abort"), new String'("abs"),
        new String'("abstract"),
        new String'("accept"), new String'("access"),
        new String'("aliased"),
        new String'("all"), new String'("and"),
        new String'("array"), new String'("at"),
        new String'("begin"), new String'("body"),
        new String'("case"), new String'("constant"),
        new String'("declare"), new String'("delay"),
        new String'("delta"), new String'("digits"),
        new String'("do"), new String'("else"),
        new String'("elsif"), new String'("end"),
        new String'("entry"), new String'("exception"),
        new String'("exit"), new String'("for"),
```

```
        new String'("function"), new String'("generic"),
        new String'("goto"), new String'("if"),
        new String'("in"), new String'("is"),
        new String'("limited"), new String'("loop"),
        new String'("mod"), new String'("new"),
        new String'("not"), new String'("null"),
        new String'("of"), new String'("or"),
        new String'("others"), new String'("out"),
        new String'("package"), new String'("pragma"),
        new String'("private"), new String'("procedure"),
        new String'("protected"),
        new String'("raise"), new String'("range"),
        new String'("record"), new String'("rem"),
        new String'("renames"),
        new String'("requeue"), new String'("return"),
        new String'("reverse"), new String'("select"),
        new String'("separate"), new String'("subtype"),
        new String'("tagged"),
        new String'("task"), new String'("terminate"),
        new String'("then"), new String'("type"),
        new String'("until"),
        new String'("use"), new String'("when"),
        new String'("while"), new String'("with"),
        new String'("xor")
    );

EOFCH : constant Character := ASCII.NUL;

procedure Feltölt_További_Betûk is
 begin
  for I in További_Betûk'Range loop
   További_Azonosító_Betû(További_Betûk(I)) := True;
  end loop;
 end;

 procedure Beolvas(Ch: out Character) is
   -- Egyszeres elõreolvasás megvalósítása.
   -- A file végén egy EOFCH karaktert ad vissza.
   -- Ez az eljárás kezeli az LNO sorszámlálót is.
   Sorkezd : Boolean := True;
 begin
   if End_Of_File(F) then
    Ch := EOFCH;
   elsif End_Of_Line(F) then
```

```
      Skip_Line(F);
      Ch := ASCII.CR;
      LNO := LNO+1;
     else
      Get(F,CH);
     end if;
   end Beolvas;

    function Kulcsszó_e(Azonosító : String)
         return Boolean is
      begin
         for I in Kulcsszavak'Range loop
           if Azonosító = Kulcsszavak(I).all
             then return True;
           end if;
         end loop;
         return False;
      end Kulcsszó_e;

      function Alfabetikus_e(X : Character)
           return Boolean is
       begin
         return (X >= 'a' and then X <= 'z') or else
                (X >= 'A' and then X <= 'Z') or else
                További_Azonosító_Betü(X);
       end Alfabetikus_e;

      function Számjegy_e(X : Character) return Boolean is
        begin
          return X >= '0' and then X <= '9';
       end Számjegy_e;

      function Alfanumerikus_e(X : Character)
           return Boolean is
       begin
        return Alfabetikus_e(X) or else Számjegy_e(X);
      end Alfanumerikus_e;

  begin
     Feltölt_További_Betûk;
     Üres_Táblázat(T);
     Put("Beolvasondó file  = ");
     Get_Line(InFile, K);  New_Line;
     Open(F, In_File, InFile(1 .. K));
```

```
    Put("Eredmény file = ");
    Get_Line(OutFile, K); New_Line;
    Create(G, Out_File, OutFile(1 .. K));

    Beolvas(Ch);
    while Ch /= EOFCH loop
        if Alfabetikus_e(Ch) then
            K := 0;
            while Ch = '_' or else Alfanumerikus_e(Ch)
              loop
               K := K + 1; ID(K) := CH; Beolvas(CH);
            end loop;
            if not Kulcsszó_e(ID(1 .. K))  and then
               not Aposztróf_Után
            then
                Bejegyez(T,ID(1 .. K),LNO);
            elsif not Kulcsszó_e(ID(1 .. K)) then
                Aposztróf_Után := False;
                if Ch = ''' then
                  Beolvas(CH);
                else
                 Bejegyez(T,ID(1 .. K),LNO);
                end if;
            end if;
        elsif Számjegy_e(Ch) then
             while Számjegy_e(Ch) loop
                   Beolvas(Ch);
             end loop;
        elsif Ch = '-' then
             Beolvas(Ch);
             if Ch = '-' then -- comment
                while Ch /= ASCII.CR loop
                      Beolvas(Ch);
                end loop;
             end if;
        elsif Ch = ''' then
            Beolvas(Ch);
            if ch /= ''' then
             Aposztróf_Után := True;
            else
              while Ch = ''' loop
                  Beolvas(Ch);
              end loop;
            end if;
```

```
            elsif Ch = '"' then
                Beolvas(Ch);
                if ch /= '"' then
                 while Ch /='"' loop
                  Beolvas(Ch);
                 end loop;
                 Beolvas(Ch);
                else
                 while Ch = '"' loop
                       Beolvas(Ch);
                 end loop;
                end if;
            else
                Beolvas(Ch);
            end if;
        end loop;
        Nyomtat(G, T);
        Close(F); Close(G);
end Xref;
```

A táblázatkezelő modul exportálja a `Táblázat` típust és a `Bejegyez` és `Nyomtat` eljárásokat. A tábla szerkezete, valamint a keresési és elérési algoritmusok rejtve maradnak a többi modul elől. A táblázatot most igényesebben ábrázoljuk, mint az előző példában: rendezett bináris fát építünk. A `Keres` eljárás jó példát mutat egy ilyen fában való keresésre, a `Fa_Bejárás` eljárás pedig egy rekurzív alprogram a fa bejárására.

9.9. Példa *(Táblázat_Osztály törzse)*

```
with Text_IO; use Text_IO;
package body Táblázat_Osztály is

  package Int_IO is new Integer_IO(Integer); use Int_IO;

  procedure Üres_Táblázat(T : in out Táblázat) is
    begin
      T := new Szó;
    end Üres_Táblázat;

  function Keres(P : Táblázat; Azonosító : String)
    return Táblázat is
     -- Megkeresi a nevet a táblázatban, a ha még nincs
     -- benne, akkor felveszi. Visszaadja a nevet
     -- tartalmazó elem címét.
       PP, Q : Táblázat; Kisebb : Boolean;
```

```
  begin PP := P; Q := PP.Jobb; Kisebb := False;
     while Q /= null loop
         PP := Q; Kisebb := Azonosító < PP.Kulcs.all;
         if Azonosító = PP.Kulcs.all then return PP; end if;
         if Kisebb then Q := PP.Bal;
           else Q := PP.Jobb;
         end if;
     end loop;
     Q := new Szó'(new String'(Azonosító),null,null,null);
     if Kisebb then PP.Bal := Q; else PP.Jobb := Q; end if;
     return Q;
  end Keres;

 procedure Bejegyez(T: in Táblázat; Azonosító: in String;
                   N: in Integer) is
         P : Táblázat;
         Q : B_Mutató;
         i : Natural;
       begin I := 0;
         P := Keres(T, Azonosító);
         Q := new Bejegyzés'(N, P.Els);
         P.Els := Q;
      end Bejegyez;

     procedure Bejegyzést_kiír(F:in File_Type;P:in Táblázat)
       is
       -- Kinyomtat egy azonosítót, s a
       -- hozzá tartozó sorszámokat.
         L : constant := 20; H : constant := 6;
         I : Integer;
         Q : B_Mutató;
     begin
        Put(F, P.Kulcs.all);
        Put(P.Kulcs.all);
        if P.Kulcs'Length <= L then
            Put(F, (1 .. L - P.Kulcs'Length => ' '));
            Put(   (1 .. L - P.Kulcs'Length => ' '));
        else
            New_Line(F);
        end if;
        I := L + H; Q := P.els;
        while Q /= null loop
          if I + H > 80 then New_Line(F);
             I := L + H;
```

```
         end if;
         Put(Q.Num, H); Put(F, Q.Num, H); I := I + H;
         Q := Q.Next;
       end loop;
       New_Line(F); Put_Line("");
     end  Bejegyzést_kiír;

   procedure Fa_Bejárás(F:in File_Type;P:in Táblázat)
      is
        begin
          if P /= null then
             Fa_Bejárás(F, P.Bal);
             Bejegyzést_kiír(F, P);
             Fa_Bejárás(F, P.Jobb);
          end if;
        end Fa_Bejárás;

   procedure Nyomtat(F: in File_Type; T: in Táblázat) is
      begin
       New_Line(F); Fa_Bejárás(F, T.Jobb);
   end Nyomtat;

end Táblázat_Osztály;
```

9.7. Feladatok

9.1. Bontsa fel a 9.1. példában bemutatott programot legalább két fordítási egységre!

9.2. Helyezzen el a 9.6. példa programszövegébe alprogramcsonkokat!

9.3. Bontsa fel a 9.8. példa programját több fordítási egységre!

10. Fejezet

Az Input-Output használata

Az input-outputot a nyelv előredefiniált csomagokon keresztül valósítja meg. Mindegyik csomag az `Ada` alapcsomag gyereke.

10.1. Külső fájlok és fájlobjektumok

A program külső környezetéből érkező, illetve oda kiírt adatokat a nyelvben úgy tekintjük, hogy azok külső fájlokat használnak. Külső fájlnak tekintünk minden olyan, a programon kívüli dolgot, amelyből értéket lehet olvasni, vagy amelybe értéket lehet írni. Egy külső fájlt egy sztring azonosít (a fájl neve). Egy másik sztring is kapcsolódhat a fájlhoz, amely a fájl rendszerfüggő jellemzőit tartalmazza (pl. hozzáférési jogok). Ennek a sztringnek a tartalmi felépítését az adott implementációban dokumentálják.

Az input-output műveleteket nem magukon a külső fájlokon definiáljuk, hanem valamely fájltípus objektumain. Ezentúl fájlon mindig egy fájl típusú objektumot értünk.

A szekvenciális fájlok input-outputját a `Sequential_IO` sablon csomag definiálja. Ha egy adott elemtípus szekvenciális input-outputját akarjuk definiálni, akkor létre kell hoznunk ennek a sablon csomagnak egy példányát, ahol ez az elemtípus az aktuális paraméter. Az eredményként kapott csomag tartalmaz egy fájltípus deklarációt, az ilyen típusú elemekből álló fájloknak (ezt nevezzük `File_Type`-nak), és az ezen fájlokra alkalmazható műveleteket - mint pl. az `Open`, `Read`, `Write` eljárásokat. A közvetlen elérésű fájlok input-outputját ugyanígy a `Direct_IO` sablon csomag definiálja. Az olvasható forma input-outputját karakteres és sztring adatokra a (nem sablon) `Text_IO`, `Wide_Character` és `Wide_String` adatokra pedig a `Wide_Text_IO` package definiálja. A különböző típusú elemeket tartalmazó fájlok input-outputjára a `Streams.Stream_IO` nem sablon csomagon keresztül van lehetőség.

Mielőtt egy input vagy output műveletet végrehajthatnánk egy fájlon, a fájlt össze kell kapcsolni egy külső fájllal. Ebben az esetben azt mondjuk,

hogy a fájlt megnyitottuk, különben pedig a fájl le van zárva. A nyelv nem definiálja, hogy mi történik a külső fájlokkal a főprogram befejezése után. Az input-output hatása a mutató típusokra is implementáció-függő. Egy megnyitott fájl mindig egy kurrens üzemmódban van, ami a következő felsorolási típusok értékei közül lehet valamelyik:

```
type File_Mode is (In_File, Inout_File, Out_File)
```

`Direct_IO` esetén, és

```
type File_Mode is (In_File, Out_File, Append_File);
```

`Sequential_IO`, `Text_IO`, `Wide_Text_IO`, és `Stream_IO` esetén. Ezek az értékek megfelelnek azoknak az eseteknek, amikor csak az olvasás, az írás és az olvasás, vagy csak az írás a végrehajtható művelet. A fájl üzemmódja változtatható.

Számos fájlművelet közös az input-output csomagokban. Ezeket együtt tárgyaljuk a szekvenciális és a direkt fájlokra is, a szöveges input-outputtal kapcsolatos további tudnivalókat pedig majd még külön írjuk le. Azokat a hibajelzéseket, amelyeket az input-output eljárások végrehajtása során kaphatunk, az `Io_Exceptions` csomag definiálja. Az ezeket kiváltó szituációkat, általában az eljárásoknál írjuk le.

Az Adában használható fájltípusokat az említett csomagok `limited private` típusként definiálják. A fájl kezelése a következő lépésekben történik:

- fájl-változó deklarálása,
- a fájl-változó összekapcsolása egy külső fájllal (fájl-nyitás),
- az olvasó, ill. író műveletek végrehajtása,
- a fájl lezárása.

Az I/O csomagok egységesek az alprogramok neveinek megválasztásában. Egy fájl létrehozása a `Create`, egy létező külső fájl megnyitása az `Open`, egy fájl lezárása a `Close` eljárással történik, bármelyik csomagot használjuk is.

10.2. Soros és közvetlen elérésű fájlok

A külső fájlokra két elérési mód definiált, a soros és a közvetlen hozzáférés. A megfelelő fájl típusokat és a hozzájuk kapcsolódó műveleteket a `Sequential_IO` és a `Direct_IO` sablon csomagok definiálják. Soros hozzáférés esetén a fájlt értékek sorozatának tekintjük, amelyek megjelenésük sorrendjében (ahogy a program vagy a környezet szolgáltatja) elérhetőek. Ha a fájlt megnyitottuk, az átvitel `In_File` és `Out_File` mód esetén a fájl elején,

`Append_File` mód esetén a fájl végén kezdődik. Közvetlen hozzáférés esetén a fájlt úgy tekintjük, mint elemek egy halmazát, amelyek lineáris sorrendben egymás utáni pozíciókat foglalnak el, bármely pozícióra vihetünk ki a fájlba egy értéket, ill. olvashatunk be egy elemet. Egy elem pozícióját az indexe mutatja, ez egy nullánál nagyobb, az implementációtól függő, `Count` típusú egész szám. Az első elem indexe 1, az utolsó elem indexe a pillanatnyi méret, ez 0, ha nincs eleme a fájlnak. A pillanatnyi méret a külső fájl egy jellemzője.

Egy megnyitott direkt fájlnak van egy kurrens indexe, ami az az index, amit a következő olvasó vagy író utasítás használni fog. Amikor a direkt fájlt megnyitjuk, ez az eljárás a kurrens indexet egyre állítja. Egy direkt fájl kurrens indexe a fájl objektum jellemzője, nem a külső fájlé.

A következő eljárások és függvények a külső fájlok vezérlésére szolgálnak, deklarációik megismétlődnek a szekvenciális és direkt input-output csomagjaiban. A szekvenciális fájloknál:

```
procedure Create ( File : in out File_Type;
                   Mode : in File_Mode:=Out_File;
                   Name: in String:="";
                   Form: in String:="");
```

a direkt fájloknál:

```
procedure Create ( File : in out File_Type;
         Mode : in File_Mode:=InOut_File;
         Name: in String:="";
         Form: in String:="");
```

Ez a művelet egy új, külső fájlt hoz létre az adott névvel és formával, és összekapcsolja ezt a külső fájlt az adott fájl objektummal. Az adott fájlt meg is nyitja. Az adott fájl elérési üzemmódja az itt megadott lesz. A kezdeti érték erre a szekvenciális és szöveges input-output esetén az `Out_File`, direkt input-outputra az `Inout_File`. Közvetlen elérés esetén a létrehozott fájl mérete az implementáció függvénye. Ha a `Name` értéke az üres jelsorozat, akkor a külső fájl a főprogram befejezése után nem hozzáférhető (ideiglenes fájl). Ha a formára az üres jelsorozatot használjuk, az azt jelenti, hogy a külső fájlra az implementáció adta kezdeti értékeket akarjuk használni.
Hibajelzések:

- `Status_Error` - ha az adott fájl már meg van nyitva,
- `Name_Error` - ha a névnél megadott jelsorozat nem használható a külső fájl azonosítására (pl. van már ilyen nevű külső fájl,
- `Use_Error` - ha a specifikált üzemmódra a környezet nem támogatja külső fájl létrehozását az adott névvel és formával.

```
procedure Open( File : in out File_Type;
                Mode : in File_Mode;
```

```
                Name: in String;
                Form: in String:="");
```

Összekapcsolja az adott fájlt a már létező, `Name`-ben adott nevű, `Form` formájú külső fájllal, és a kurrens üzemmódot a `Mode`-ban megadottra állítja. A fájl „nyitott" lesz.

Hibajelzések:

- `Status_Error` - ha az adott fájl már meg van nyitva,
- `Name_Error` - ha nincs a `Name`-ben megadott nevű külső fájl,
- `Use_Error` - ha a specifikált üzemmódra a környezet nem támogatja ilyen nevű és formájú külső fájl megnyitását (pl. nem megfelelő üzemmódot akarok, v. hibás a form rész, v. nincs jogom írásra stb.).

```
procedure Close ( File : in out File_Type ) ;
```

Megszünteti a kapcsolatot az adott fájl és a hozzá kapcsolódó külső fájl között. Az adott fájl le lesz zárva.

Hibajelzések:

- `Status_Error` - ha az adott fájl nincs megnyitva.

```
procedure Delete( File : in out File_Type);
```

Törli az adott fájllal összekapcsolt külső fájlt. Az adott fájlt lezárja, a külső fájl megszűnik.

Hibajelzések:

- `Status_Error` - ha az adott fájl nincs megnyitva,
- `Use_Error` - ha a környezet nem támogatja a külső fájl törlését.

```
procedure Reset(File : in out File_Type;
                Mode : in File_Mode);
procedure Reset(File : in out File_Type);
```

Visszaállítja az adott fájlt, hogy a belőle való olvasást, vagy a bele való írást `In_File`, `Out_file` és `InOut_File` módra nyitott fájlok esetén az elejéről, `Append_File` estén a végén kezdjük. Közvetlen hozzáférés esetén ez azt jelenti, hogy a kurrens indexet 1-re állítja. Ha `Mode` paraméter is szerepel, akkor a kurrens üzemmódot is változtatja.

Hibajelzések:

- `Status_Error` - ha az adott fájl nincs megnyitva,
- `Use_Error` - ha a környezet nem támogatja a külső fájlra a visszaállítást, vagy a specifikált üzemmódot.

```
function Mode ( File : in File_Type ) return File_Mode;
```

Megadja az adott fájl kurrens üzemmódját.
Hibajelzések:
`Status_Error` - ha az adott fájl nincs megnyitva.

```
function  Name( File : in  File_Type) return String;
```

A visszaadott jelsorozat egyértelműen azonosítja az adott fájllal pillanatnyilag összekapcsolt külső fájl nevét, (így pl. használható `Open` utasításban). Ha a környezet megengedi a névnek alternatív specifikációit (pl. rövidítéseket), a visszaadott jelsorozat a név teljes specifikációjának felel meg.
Hibajelzések:
`Status_Error` - ha az adott fájl nincs megnyitva,
`Use_Error` - ha az objektumhoz kapcsolt fájl ideiglenes, azaz névvel nem azonosítható.

```
function  Form( File : in  File_Type) return String;
```

Visszaadja az adott fájllal pillanatnyilag összekapcsolt külső fájl formajelsorozatát. Ha a környezet megengedi a forma alternatív specifikációit (pl. rövidítéseket), az itt visszaadott jelsorozat akkor is a forma teljes specifikációjának felel meg.
Hibajelzések:
`Status_Error` - ha az adott fájl nincs megnyitva.

```
function Is_Open(  File : in  File_Type)  return Boolean;
```

`True` értéket ad, ha a fájl meg van nyitva (vagyis össze van kapcsolva egy külső fájllal), különben `False`-t.

10.2.1. Szekvenciális fájlok

A továbbiakban először a szekvenciális fájlok iró-olvasó műveleteit tekintjük át. Ezekben a műveletekben mindig a `Status_Error` hibajelzés lép fel, ha az adott fájl nincs előzőleg megnyitva.

```
procedure Read( File : in  File_Type;
                Item : out Element_Type);
```

`In_File` üzemmódban lévő fájlon használható. Kiolvas egy elemet az adott fájlból, és az `Item` paraméterben adja vissza.
Hibajelzések:
`Mode_Error` - ha a fájl nincs `In_File` üzemmódban,
`End_Error` - ha nem lehet több elemet kiolvasni az adott fájlból,

`Data_Error` - ha az olvasott elem nem értelmezhető `Element_Type` típusú értékként. (Megengedett, hogy egy implementáció ne végezze el ezt az ellenőrzést, ha az túl bonyolult.)

```
procedure Write( File : in  File_Type;
                 Item : in Element_Type);
```

`Out_File` vagy `Append_File` üzemmódban lévő fájlon használható. Kiírja az `Item` paraméter értékét az adott fájlba.
Hibajelzések:

`Mode_Error` - ha a fájl nincs `Out_File` vagy `Append_File` üzemmódban,

`Use_Error` - ha a külső fájl kapacitása kimerült.

```
function End_Of_File ( File : in  File_Type ) return Boolean;
```

`In_File` üzemmódban lévő fájlon használható. `True` értéket ad, ha nem olvasható több elem az adott fájlból, különben `False`.
Hibajelzések:

`Mode_Error` - ha a fájl nincs `In_File` üzemmódban.

10.2.2. Közvetlen elérésű fájlok

A következőkben a direkt fájlok író és olvasó műveleteit ismertetjük. Itt is mindig `Status_Error` hibajelzés lép fel, ha az adott fájl nincs előzőleg megnyitva.

```
procedure Read( File : in  File_Type;
                Item: out Element_Type) ;
procedure Read( File : in  File_Type;
                Item: out Element_Type
                From : in Positive_Count) ;
```

`In_File` vagy `Inout_File` üzemmódban lévő fájlon használható. A második esetben az adott fájl kurrens indexét a `From` paraméter által megadott indexértékre állítja. Ezután mindkét forma esetén az `Item` paraméterben visszaadja a fájl azon elemét, amelynek pozícióját a fájl kurrens indexe mutatja, s eggyel növeli a kurrens indexet.
Hibajelzések:

`Mode_Error` - ha a fájl `Out_File` üzemmódban van,

`End_Error` - ha a használt index nagyobb a külső fájl méreténél,

`Data_Error` - ha az olvasott elem nem értelmezhető `Element_Type` típusú értékként. (Megengedett, hogy egy implementáció ne végezze el ezt az ellenőrzést, ha az túl bonyolult.)

```
procedure Write( File : in  File_Type; Item: in Element_Type);
procedure Write( File : in  File_Type; Item: in Element_Type;
                 To : in Positive_Count);
```

Inout_File vagy Out_File üzemmódban lévő fájlon használható. A második forma esetén az adott fájl kurrens indexét a To paraméter értékére állítja. Ezután mindkét forma esetén az Item paraméter értékét adja a fájl azon elemének, amelynek pozícióját a fájl kurrens indexe kijelölte, majd eggyel növeli a kurrens indexet.
Hibajelzések:

Mode_Error - ha a fájl In_File üzemmódban van,

Use_Error - ha a külső fájl kapacitása kimerült.

```
procedure Set_Index( File : in  File_Type;
                     To : in Positive_Count);
```

Tetszőleges üzemmódban lévő fájlon végrehajtható. Az adott fájl kurrens indexét az adott indexértékre állítja (ez lehet nagyobb a fájl pillanatnyi méreténél)

```
function Index( File : in  File_Type) return Positive_Count;
```

Tetszőleges üzemmódban lévő fájlon végrehajtható. Az adott fájl kurrens indexét adja vissza.

```
function Size(File : in  File_Type) return Count ;
```

Tetszőleges üzemmódban lévő fájlon végrehajtható. Az adott fájllal összekapcsolt külső fájl kurrens méretét adja.

```
function End_Of_File(File : in File_Type) return Boolean;
```

In_File vagy InOut_File üzemmódban lévő fájlon használható. True értéket ad, ha a kurrens index nagyobb a külső fájl méreténél, False-t különben.
Hibajelzések:

Mode_Error - ha a fájl Out_File üzemmódban van.

A következő program egy egyszerű példa a Direct_IO használatára.

```
with Direct_Io;
procedure DIOpélda is

   package Dio is new Direct_Io(Character); use Dio;
   F: Dio.File_Type;
```

```
begin
   Create(f,name=>"példa.dat");
   for J in Integer range 1..1000 loop
         Write(f,'a');
   end loop;

   for J in reverse Positive_Count Range 1..250 loop
      Write(F,'l', J*4-2);
   end loop;

   for J in Positive_Count Range 1..250 loop
      Set_Index(F, J*4-1);
      Write(F,'m');
   end loop;

   Close(f);
end DIOpélda;
```

A program elején példányosítjuk a `Direct_IO` sablon csomagot, jelen esetben `Character` típust adunk át paraméterként. Ennek hatására létrejönnek a `Character` elemekből álló fájlok kezeléséhez szükséges műveletek (`Create`, `Open` stb.) és típusok (`File_Type`).

A program három ciklust tartalmaz. Az első ciklus 1000 db „a" betűt ír egy fájlba, itt szekvenciálisan használtuk a fájlt. A második ciklus az előző ciklus által feltöltött fájl végétől indulva, és a fájl eleje felé haladva felülír karaktereket. A harmadik ciklus funkciójában hasonlít a másodikhoz, bár az adatokat a fájl eljétől kezdi el átírni és az indexet nem a write műveletnek adja meg, hanem a `Set_Index` parancsot használja.

A program végeredménye egy `példa.dat` nevű fájl, mely 250 db „alma" sztringet tartalmaz.

10.3. A Storage_IO csomag

A sablon `Storage_IO` csomag az egy memóriabufferen történő irást-olvasást valósítja meg. A csomag specifikációja az alábbi:

```
with Ada.IO_Exceptions;
with System.Storage_Elements;
generic
 type Element_Type is private;
package Ada.Storage_IO is
 pragma Preelaborate(Storage_IO);
```

```
  Buffer_Size : constant System.Storage_Elements.Storage_Count
     := implementation-defined;
  subtype Buffer_Type is
     System.Storage_Elements.Storage_Array(1..Buffer_Size);

  procedure Read (Buffer : in  Buffer_Type;
                  Item : out Element_Type);

  procedure Write(Buffer : out Buffer_Type;
                  Item : in  Element_Type);

  Data_Error : exception renames IO_Exceptions.Data_Error;
end Ada.Storage_IO;
```

A csomag minden példányában a Buffer_Size konstans értéke annak a buffernek a mérete, amekkorára egy Element_Type típusú objektum – az implementáció általi – belső tárolásához szükséges.

A csomag Read és Write eljárásai a Direct_IO csomag Read és Write eljárásaival analógok, csak az adott elemet egy külső fájl helyett a megadott memóriabufferből olvassa, illetve oda írja ki.

A Storage_IO csomagban használt buffer egyszerre csak egy elemet tartalmaz, míg a Direct_IO-ban használt külső fájl elemek egy sorozatát tartalmazza.

10.4. Textfájlok kezelése

A Text_IO csomag teszi lehetővé az olvasható formák input-outputját. Minden fájlt szekvenciálisan lehet olvasni, ill. minden fájlra szekvenciálisan lehet írni; a karakterek sorozatát sorokká, a sorok sorozatát oldalakká csoportosítjuk.

Az előzőekben megadott fájlkezelési lehetőségek a szöveges input-outputra is alkalmazhatóak. A Read és Write eljárások helyett azonban Get és Put eljárások használhatók, amelyek a megfelelő típusú értékeket behozzák a szövegfájlokból, ill. ilyeneket továbbítanak a fájlba.

Számos Get és Put eljárás létezik az Item paraméter különböző típusaira. Minden szövegfájlra alkalmazható Get és Put eljárásnak , és sok más alprogramnak is, két formája van: az egyik tartalmaz fájl-paramétert, a másik nem. Minden ilyen Get eljárás egy input fájlra, a Put egy output fájlra vonatkozik. Ha nincs külön fájl paramétere, akkor alapértelmezett input, ill. output fájlt használ, ezek az ún. aktuális (kurrens) input, ill. output fájlok (a Current_Input és Current_Output függvényekkel lekérdezhetők).

A program indulásának pillanatában ezek az ún. standard text-fájlok, amelyeket a futtató rendszer automatikusan megnyit. Interaktív rendsze-

rekben ezek a fájlok a terminálhoz vannak rendelve: a klaviatúra a standard input, a képernyő a standard output fájl. Az aktuális fájl az adott programra nézve átirányítható más text-fájlra is a `Set_Input` és `Set_Output` eljárásokkal. Ekkor a `Get` és `Put` paraméter nélküli alakja az így átirányított fájlra vonatkozik. (Visszairányítás pl. `Set_Input(Standard_Input)` segítségével.)

Logikai szempontból egy szövegfájl oldalak sorozata, egy oldal sorok sorozata, és egy sor karakterek sorozata. A sorok végét sorvégjel, míg az oldalak végét egy sorvégjel és egy őt követő lapvégjel jelenti. A fájl végén egy sorvégjel, egy lapvégjel és egy fájlvégjel található.

A fenti végjelek az egyes implementációkban különbözhetnek, sőt nem is feltétlenül vannak speciális karakterekkel vagy karaktersorozatokkal megvalósítva. Ezzel persze a csomag felhasználójának nem kell törődnie, hiszen a logikai struktúrára akkor is támaszkodhat.

A karakterek a sorokon belül 1-gyel kezdve vannak megszámozva. A sorvégjel sorszáma eggyel nagyobb, mint a sorban található karakterek száma. A sorok az oldalakon belül, ill. az oldalak a fájlon belül ugyanezen rendszer szerint vannak megszámozva. Az aktuális karakter sorszáma (oszlopszám) a következő karakter sorszáma. Az aktuális sor sorszám az aktuális sor sorszáma, míg az aktuális lapszám az aktuális lap sorszáma. Ezek a számok a `Count` típus `Positive_Count` altípusába esnek.

```
type Count is range 0 .. implementation-defined;
subtype Positive_Count is Count range 1 .. Count'Last;
```

Egy output fájlra beállítható és lekérdezhető a maximális sorhossz, ill. oldalhossz. Ha a kiírandó érték nem fér bele a feldolgozás alatt álló sorba egy specifikált maximális sorhossz mellett, akkor automatikusan új sort kezd, mielőtt végrehajtaná az átvitelt, ugyanígy, ha egy új sor nem fér rá a feldolgozás alatt álló oldalra egy specifikált maximális oldalhossz mellett, akkor automatikusan új oldalt kezd. Amikor egy fájlt `Out_File` vagy `Append_File` üzemmódban nyitunk meg, mindkét érték 0, ez megegyezés szerint azt jelenti, hogy nincs rájuk korlát. (Következésképpen az output egyetlen sorból áll, ha nem használjuk explicite a `New_Line`, ill. `New_Page` parancsokat!)

A szövegfájlok megnyitási módjai, inicializáló és bezáró műveletei pontosan ugyanolyanok, mint a szekvenciális fájlok esetén, de bizonyos további hatásaik is vannak a ottaniakhoz képest.

A `Create` és `Open` eljárásokban: ha a fájlt `Out_File` vagy `Append_File` üzemmódra nyitottunk meg, az oldalhossz és sorhossz nem korlátozott; mindhárom üzemmód esetén az aktuális oszlop, sor és oldal számot 1-re állítja. `Append_File` mód esetén implementációfüggő, hogy az eredeti szöveg és az új szöveg között lesz-e oldalvégjel.

A `Close` eljárásban: `Out_File` és `Append_File` üzemmódú fájl esetén még egy `New_Page` eljárást hív (kivéve, ha az aktuális oldal éppen befejeződött), majd kiír egy fájlvég jelet. Hasonlóan a `Reset` eljárás `Out_File` és

Append_File üzemmódú fájl esetén még egy New_Page eljárást hív (kivéve, ha az aktuális oldal éppen befejeződött) majd kiír egy fájlvég jelet. Ha Out_File üzemmódúra állítjuk a fájlt, az oldalhossz és sorhossz nem korlátozott; mindkét üzemmód esetén az aktuális oszlop, sor és oldal számot 1-re állítja. Append_File mód esetén implementációfüggő, hogy az eredeti szöveg és az új szöveg között lesz-e oldalvégjel. Mode_Error hibajelzés lép fel, ha a kurrens default input, ill. output fájl üzemmódját akarjuk változtatni.

A továbbiakban először a default fájlokat kezelő műveleteket tekintjük át. Ezeket a default fájlokat a fájl paraméter nélküli Get és Put műveletek használják.

```
procedure Set_Input(File : in File_Type);
```

In_File üzemmódban lévő fájlon használható. A kurrens default input fájl az itt megadott lesz.
Hibajelzések:
Mode_Error - ha a fájl nincs In_File üzemmódban
Status_Error - ha az adott fájl nincs megnyitva

```
procedure Set_Output(File : in File_Type);
procedure Set_Error (File : in File_Type);
```

Out_File vagy Append_File üzemmódban lévő fájlon használható. A kurrens default output fájl (illetve a kurrens default hibafájl) az itt megadott lesz.
Hibajelzések:
Mode_Error - ha a fájl nincs Out_File vagy Append_File üzemmódban
Status_Error - ha az adott fájl nincs megnyitva

```
function Standard_Input return File_Type;
function Standard_Input return File_Access;

function Standard_Output return File_Type;
function Standard_Output return File_Access;

function Standard_Error return File_Type;
function Standard_Error return File_Access;
```

Megadják a rendszer által definiált standard input, output és hibafájlokat, illetve a rájuk mutató mutatókat.

```
function Current_Input return File_Type;
function Current_Input return File_Access;
```

```
function Current_Output return File_Type;
function Current_Output return File_Access;

function Current_Error return File_Type;
function Current_Error return File_Access;
```

Megadják az aktuális standard input, output és hibafájlokat, illetve a rájuk mutató mutatókat.

```
procedure Flush (File : in out File_Type);
procedure Flush;
```

A `Flush` eljárások hatása megegyezik a `Streams.Stream_IO` csomagban található azonos nevű eljárásokéval.

A `Get` és a `Put` eljárásokat megelőlegezve lássuk a következő példaprogramot, mely a standard szövegfájlok átirányítását mutatja be.

10.1. Példa *(Standard szövegfájok átirányítása)*

```
with Text_Io; use Text_Io;
procedure Átirányít is
   F: File_Type;
   I: Character;
begin
   Create( F, Name =>"példa.txt");
   Set_Output(F);
   Put("Ez a szöveg most nem a képernyőre,");
   Put("hanem a példa.txt fájlba került.");
   New_Line;
   Set_Output(Standard_Output);
   Put_Line("A példa.txt fájl tartalma:");

   Reset(F,In_File);
   Set_Input(F);

   while not End_Of_File loop
      while not End_Of_Line loop
         Get(I);
         Put(I);
      end loop;
      Skip_Line;
      New_Line;
   end loop;
   Close(F);
end Átirányít;
```

A programban először létrehozunk egy `példa.txt` nevű fájlt, majd ezt állítjuk be kurrens default output fájlnak, amibe a `Put` és a `New_Line` műveletek használatával adatokat írunk. Ezután a kurrens default input fájlnak a `példa.txt` fájlt jelöljük meg, a kurrens default output fájlt pedig visszaállítjuk a standard outputra, és kiírjuk a `példa.txt` fájl tartalmát.

10.4.1. Sor- és oldalhosszok beállítása és lekérdezése

Az alábbiakban azokat a műveleteket vizsgáljuk meg, amelyek az `Out_File` vagy `Append_File` üzemmódban lévő fájlok sor- és lapszerkezetének kezelésére szolgálnak. A műveletek azon változatai, ahol fájl paraméter nem szerepel, mindig az aktuális default output fájlra vonatkoznak.

Minden esetben `Status_Error` hibajelzés lép fel, ha az adott fájl nincs megnyitva, és `Mode_Error`, ha a fájl nincs `Out_File` vagy `Append_File` üzemmódban.

```
procedure Set_Line_Length(File : in File_Type;
                          To : in Count);
procedure Set_Line_Length(To : in Count);
```

A maximális sorhosszt a `To` paraméterben megadottra állítja. (`To = 0` korlátozás nélkülit jelent!) A sorhossz beállítása nincs hatással a fájl eddigi tartalmára, csak a beállítás utáni kiírásokra vonatkozik.
Hibajelzések:

`Use_Error` - ha a specifikált sorhossz nem felel meg a fájllal összekapcsolt külső fájlnak.

```
procedure Set_Page_Length(File : in File_Type;
                          To : in Count);
procedure Set_Page_Length(To : in Count);
```

A maximális oldalhosszt a `To` paraméterben megadottra állítja. (`To = 0` itt is korlátozás nélkülit jelent!)
Hibajelzések:

`Use_Error` - ha a specifikált oldalhossz nem felel meg a fájllal összekapcsolt külső fájlnak.

```
function Line_Length(File : in File_Type) return Count;
function Line_Length return Count;
```

Visszaadja a maximális sorhosszt a specifikált output fájlra, 0-t , ha nem korlátozott.

```
function Page_Length(File : in File_Type) return Count;
function Page_Length return Count;
```

Visszaadja a maximális sorhosszt a specifikált output fájlra, 0-t , ha nem korlátozott.

10.4.2. Oszlopok, sorok, és oldalak műveletei

Az alábbi műveletek a sor- és lapszerkezet explicit kezelésére adnak lehetőséget. A műveletek azon változatai, ahol fájl paraméter nem szerepel, mindig a megfelelő aktuális default input vagy output fájlra vonatkoznak. Minden esetben `Status_Error` hibajelzés lép fel, ha az adott fájl nincs megnyitva.

```
procedure New_Line(File : in File_Type;
                   Spacing : in Positive_Count := 1);
procedure New_Line(Spacing : in Positive_Count := 1);
```

`Out_File` vagy `Append_File` üzemmódban lévő fájlokra használható. Kiír egy sorvége jelet, az aktuális oszlop száma 1-re áll, `Spacing`-gel nő az aktuális sor száma, kivéve, ha ezzel esetleg túllépné a maximális lapméretet, akkor megfelelően tördel, új lapot kezd, és ott a megfelelő sorra áll.
Hibajelzések:
`Mode_Error` - ha a fájl nincs `Out_File` vagy `Append_File` üzemmódban

```
procedure Skip_Line(File : in File_Type;
        Spacing : in Positive_Count := 1);
procedure Skip_Line(Spacing : in Positive_Count := 1);
```

`In_File` üzemmódban lévő fájlokra használható. Ha `Spacing` értéke 1, megkeresi az aktuális sor végét, aktuális oszlopszám 1-re áll, ha nem oldalvége jel a következő, akkor az aktuális sor száma 1-gyel nő, különben átugrik az oldal vége jelen, az aktuális oldalszámot növeli, és az aktuális sor száma 1 lesz. Egynél nagyobb `Spacing` esetén az előzőeket ismétli `Spacing`-szer.
Hibajelzések:
`Mode_Error` - ha a fájl nincs `In_File` üzemmódban
`End_Error` - ha fájl végét akar olvasni

```
function End_Of_Line(File : in File_Type) return Boolean;
function End_Of_Line return Boolean;
```

`In_File` üzemmódban lévő fájlokra használható. `True` értéket ad, ha a következő jel sor vége, vagy fájl vége.
Hibajelzések:
`Mode_Error` - ha a fájl nincs `In_File` üzemmódban

```
procedure New_Page(File : in File_Type);
procedure New_Page;
```

Out_File vagy Append_File üzemmódban lévő fájlokra használható. Kiír egy sorvége jelet, ha az aktuális sor még nincs befejezve, vagy ha az aktuális oldal üres, majd egy oldal vége jelet. Az aktuális sor és oszlop száma 1-re áll, az aktuális oldal száma 1-gyel nő.
Hibajelzések:
Mode_Error - ha a fájl nincs Out_File vagy Append_File üzemmódban

```
procedure Skip_Page(File: in File_Type);
procedure Skip_Page;
```

In_File üzemmódban lévő fájlokra használható. Átugrik minden karaktert és sor vége jelet, amíg az első oldal vége jelet meg nem találja. Ekkor az aktuális sor és oszlop száma 1-re áll, az aktuális oldalszámot 1-gyel növeli.
Hibajelzések:
Mode_Error - ha a fájl nincs In_File üzemmódban
End_Error - ha fájl végét akar olvasni

```
function End_Of_Page(File : in File_Type) return Boolean;
function End_Of_Page return Boolean;
```

In_File üzemmódban lévő fájlokra használható. True értéket ad, ha a következő jel sor vége, lap vége pár, vagy fájl vége.
Hibajelzések:
Mode_Error - ha a fájl nincs In_File üzemmódban

```
function End_Of_File(File : in File_Type) return Boolean;
function End_Of_File return Boolean;
```

In_File üzemmódban lévő fájlokra, True, ha a következő jel fájl vége, vagy sor-, oldal- és fájlvég kombináció.
Hibajelzések:
Mode_Error - ha a fájl nincs In_File üzemmódban

A következő eljárások az írás vagy olvasás aktuális pozícióját vezérlik. A fájl paraméter nélküli műveletek az aktuális default output fájlra vonatkoznak.

```
procedure Set_Col(File : in File_Type;
                  To : in Positive_Count);
procedure Set_Col(To : in Positive_Count);
```

Ha a fájl Out_File vagy Append_File üzemmódú: ha a To-ban specifikált érték nagyobb az aktuális oszlopszámnál, akkor szóközöket ír ki, és egyesével növeli az aktuális oszlopszámot, míg egyenlő nem lesz a To-ban megadott értékkel. Ha a To-ban specifikált érték egyenlő az aktuális oszlopszámmal, akkor nincs hatása, ha pedig ha a To-ban specifikált érték kisebb az aktuális oszlopszámnál, akkor New_Line-t hív, majd To-1 helyközt hagy ki, és az aktuális oszlopszám egyenlő lesz a To-ban megadott értékkel.
Hibajelzés:

Layout_Error - ha van megadva egy (nem 0) maximális sorhossz, és a To-beli érték ennél nagyobb.

Ha a fájl In_File üzemmódú: Átugrik minden karaktert és sor vége, ill. oldal vége jelet, amíg a következő olvasandó karakter aktuális oszlop száma a To-ban specifikált nem lesz. Nincs hatása, ha az aktuális oszlop száma éppen To. Minden karakter, ill. elhatárolójel átvitele megfelelően állítja az aktuális oszlop-, sor- és oldalszámokat. (A rövid sorokat átugorja, míg nem talál egy olyan sort, aminek van a megfelelő oszlopon karaktere.)
Hibajelzések:

End_Error - ha fájl végét akar olvasni

```
procedure Set_Line(File : in File_Type;
                   To : in Positive_Count);
procedure Set_Line(To : in Positive_Count);
```

Ha a fájl Out_File vagy Append_File üzemmódú: ha a To-ban specifikált érték nagyobb az aktuális sor számánál, akkor New_Line ismételt hívása, míg az aktuális sorszám egyenlő nem lesz a To-ban megadott értékkel. Ha a To-ban specifikált érték megegyezik az aktuális sorszámmal, akkor nincs hatása, ha pedig ha a To-ban specifikált érték kisebb az aktuális sor számánál, akkor New_Page-et hív, majd New_Line-t (To-1) Spacing értékkel és az aktuális sorszám egyenlő lesz a To-ban megadott értékkel.
Hibajelzés:

Layout_Error - ha van megadva (nem 0) maximális oldalhossz, és a To-beli érték ennél nagyobb.

Ha a fájl In_File üzemmódú: Skip_Line ismételt hívása (Spacing = 1 értékkel) mindaddig, amíg az aktuális sor száma a To-ban specifikált nem lesz. Nincs hatása, ha az aktuális sor száma éppen To. Minden karakter, ill. elhatárolójel átvitele megfelelően állítja az aktuális oszlop-, sor- és oldalszámokat. (A rövid oldalakat átugorja, amíg nem talál olyan oldalt, amelynek van a megfelelő sorszámú sora.)
Hibajelzések:

End_Error - ha fájl végét akar olvasni

```
function Col(File : in File_Type) return Positive_Count;
function Col return Positive_Count;
```

Az aktuális oszlop számát adja vissza.

Hibajelzés:

`Layout_Error` - ha a visszaadott érték nagyobb, mint a `Count` típus felső határa.

```
function Line(File : in File_Type) return Positive_Count;
function Line return Positive_Count;
```

Az aktuális sor számát adja vissza.

Hibajelzések:

`Layout_Error` - ha a visszaadott érték nagyobb, mint a `Count` típus felső határa.

```
function Page(File : in File_Type) return Positive_Count;
function Page return Positive_Count;
```

Az aktuális oldal számát adja vissza.

Hibajelzések:

`Layout_Error` - ha a visszaadott érték nagyobb, mint a `Count` típus felső határa.

Az oszlop-, sor- vagy oldalszám lehet nagyobb, mint a `Count'Last`, ez csak akkor vált ki kivételt, ha le akarjuk kérdezni.

Az oszlopokra, sorokra és oldalakra vonatkozó műveleteket demonstrálja a következő program:

10.2. Példa *(Oszlop- és sorműveletek)*

```
with text_io; use text_io;
procedure Oldalak is
   F: file_type;
   I: Character;
begin
   Create(f,name=>"példa.txt");
   Set_Line_Length(F,20);
   Set_Page_Length(F,3);
   for I in Integer range 1..234 loop
      Put(F,'a');
   end loop;

   Reset(F,mode=>in_file);
   while not End_Of_File(F) loop
```

```
      if End_Of_Line(F) then
         Put(" Sor vége");
         New_Line;
      end if;
      if End_Of_Page(F) then
         Put("Lap vége");
         New_Line;
      end if;
      Get(F,I);
      Put(I);
   end loop;
   New_Line;
   Put_line("File vége");

   Close(F);
end Oldalak;
```

A program létrehoz egy példa.txt nevű fájlt, majd miután beállítja a sorokra és lapokra jellemző tulajdonságokat kiír 234 „a" betűt a külső fájlba. Ennek befejezése után a program a standard outputra – ez általában a képernyő – írja a példa.txt fájl tartalmát, jelezve a sorok és az oldalak végét.

A program második részében kihasználtuk a Get művelet azon tulajdonságát, hogy csak látható karaktert ad vissza – ha újsor vagy lapvége karaktert olvasna, akkor továbblép a következő karakterre. A Get művelet eme tulajdonságairól a következő oldalak írnak.

10.4.3. Get és Put eljárások

A Get és Put eljárásokat a Character, String, numerikus és felsorolási típusokra készítették el, közös jellemzőiket együtt tárgyaljuk. A Character és String típusú elemek Get és Put eljárásai egyedi karakter-értékeket kezelnek, a numerikus és felsorolási típusokra vonatkozó Get eljárások lexikai elemként analizálják a bejövő karakter-sorozatot, a Put eljárások pedig megfelelő lexikális elemként továbbítják az adott értékeket.

Minden Get, ill. Get_Line egy In_File üzemmódban és minden Put, ill. Put_Line egy Out_File vagy Append_File üzemmódban lévő fájlra értelmes, megfelelően módosítják a specifikált fájl aktuális oszlop, sor és lap számát.

A Get, Get_Line, Put, és Put_Line műveletek Status_Error kivételt váltanak ki, ha meg nem nyitott fájlra hívjuk meg őket. Mode_Error hibát jeleznek a Get és Get_Line műveletek ha a fájl nem In_File üzemmódú, illetve a Put és Put_Line műveletek, ha a fájl üzemmódja nem Out_File vagy Append_File.

A karakter típusra az alábbi eljárásokat használhatjuk:

```
procedure Get(File : in File_Type; Item : out Character);
procedure Get(Item : out Character);
```

A sor- és lapvége jeleket átugorva olvassa a következő karaktert az adott input fájlból, és az `Item` paraméter értékének adja.
Hibajelzés:
`End_Error` - ha fájl végén túl akar olvasni

```
procedure Put(File : in File_Type; Item : in Character);
procedure Put(Item : in Character);
```

Ha a aktuális oszlopszám nagyobb, mint a specifikált maximális sorhossz (ha van ilyen megadva), akkor az adott karakter kiírása előtt `New_Line`-t hív (`Spacing = 1` paraméterrel).

```
procedure Look_Ahead (File  : in  File_Type;
                      Item  : out Character;
                      End_Of_Line : out Boolean);
procedure Look_Ahead (Item  : out Character;
                       End_Of_Line : out Boolean);
```

Az `End_Of_Line` paramétert `True`-ra állítja, ha a következő jel sor-, lap- vagy fájlvég jel (ekkor `Item` értéke nem definiált), ellenkező esetben `End_Of_File` értékét `False`-ra állítja, és `Item`-ben visszaadja a következő karaktert anélkül, hogy azt a fájlból kiolvasná. `Mode_Error` kivételt vált ki, ha a fájl nem `In_File` üzemmódú.

```
procedure Get_Immediate(File : in  File_Type;
        Item : out Character);
procedure Get_Immediate(Item : out Character);
```

Beolvassa a következő karaktert, legyen az vezérlő vagy grafikus karakter. Az aktuális oszlop, sor és lapszámot nem változtatja meg. `Mode_Error` kivételt vált ki, ha a fájl nem `In_File` üzemmódú, és `End_Error`-t a fájl végén.

```
procedure Get_Immediate(File  : in  File_Type;
                        Item  : out Character;
                        Available : out Boolean);
procedure Get_Immediate(Item  : out Character;
                        Available : out Boolean);
```

Ha van még beolvasható karakter (vezérlő vagy grafikus) a fájlban, akkor azt `Item`-be beolvassa, és `Available`-t `True`-ra állítja. Ellenkező esetben `Available` értéke `False`, `Item` értéke pedig nem definiált lesz. Az aktuális oszlop, sor és lapszámot nem változtatja meg. `Mode_Error` kivételt vált ki, ha a fájl nem `In_File` üzemmódú, és `End_Error`-t a fájl végén.

A fenti `Get_Immediate` műveletekkel lehetőségünk van pl. a billentyűzet „azonnali" olvasására. Az `Available` paraméter nélküli eljárás ebben az esetben vár, amíg lesz beolvasható karakter.

`String` típusra az alábbi műveleteket használhatjuk:

```
procedure Get(File : in File_Type; Item : out String);
procedure Get(Item : out String);
```

Meghatározza az adott sztring hosszát, és ennyiszer hívja a karakterek `Get`-jét.
Hibajelzés:
`End_Error` - ha fájl végén túl akar olvasni

```
procedure Put(File : in File_Type; Item : in String);
procedure Put(Item : in String);
```

Meghatározza az adott sztring hosszát, és ennyiszer hívja a karakterek `Get`-jét.

```
procedure Get_Line(File : in File_Type;
                   Item : out String;
                   Last : out Natural);
procedure Get_Line(Item : out String;
                   Last : out Natural);
```

Sor végéig beolvas a fájlból a sztringbe, ezután `Skip_Line`-t hív, és befejezi az olvasást. Ha a sztring végéhez ér, akkor is befejezi az olvasást. A `Last`-ban megadja az utolsó beolvasott elem indexét. Ha nincs beolvasott karakter, akkor a `Last`-ban az `Item'First`-nél eggyel kisebb értéket ad vissza. A `Last` pozíció utáni karakterek (ha a sztring nem lett feltöltve) értéke nem definiált.
Hibajelzés:
`End_Error` - ha fájl végén túl akar olvasni

```
procedure Put_Line(File : in File_Type; Item : in String);
procedure Put_Line(Item : in String);
```

Meghívja a `Put` műveletet a sztringre, majd végrehajt egy `New_Line` eljárást `Spacing = 1` értékkel.

A következő példán bemutatjuk az előbbi műveletek egy részének használatát.

10.3. Példa *(A* `Get` *és* `Put` *használata)*

```
with Text_Io; use Text_Io;
procedure Get_Put is
   F: File_Type;
   I: Character;
   B: Boolean := FALSE;
begin
      -- példa.txt létrehozása
   Create(F, Name=>"példa.txt");

      -- a standard inputról olvasottak
      -- a példa.txt-be másolása
   while not End_Of_File loop
      while not End_Of_Line loop
         Get(I);
         Put(F, I);
      end loop;
      Skip_Line;
      New_Line(F);
   end loop;
   Close(F);

      -- a példa.txt kiratása
   Put_Line("A példa.txt tartalma:");
   Open(F, Name=>"példa.txt", Mode=>In_File );
   while not End_Of_File(F) loop
      Put_Line(Get_Line(F));
   end loop;
   Close(F);

      -- billentyűlenyomásra várakozás
   Put("Nyomjon le egy billentyűt,");
   Put_Line(" hogy a program befejeződjön!");
   while not B loop
      Get_Immediate(Item=>I, Available=>B);
   end loop;
   New_Line;
end Get_Put;
```

A program indulásakor létrehoz egy **`példa.txt`** nevű fájlt, amibe a standard inputról olvasott adatokat másolja. Ezt egészen a fájl vége jelig teszi – billentyűzet esetén `UNIX` rendszereken egy *újsor* karakter után a `^d`, `DOS` estén egy *újsor* karakter után a `^z` jelenti a fájl végét.

A program kilépés előtt egy tetszőleges (nem funkció) billentyű lenyomására vár.

10.4.4. Egész típusok input-outputja

Az `Integer_IO` és `Modular_IO` beágyazott sablon csomagok valósítják meg az előjeles és moduláris egész típusok input-outputját; példányosítani kell őket használat előtt a megfelelő egész típusra . Az alábbi leírásban `Num` jelöli a sablon paraméter típust.

A csomagban a számok kiíratási szélességét vezérlő default érték a megfelelő egész típus szélessége:

```
subtype Field  is Integer range 0 .. implementációfüggő;
Default_Width : Field := Num'Width;
```

Az értékeket decimális, vagy más alapú számként kezelhetjük, ennek default értéke 10:

```
subtype Number_Base is Integer range 2 .. 16;
 Default_Base : Number_Base := 10;
```

A csomagban az alábbi eljárások találhatóak:

```
procedure Get(File : in File_Type;
              Item : out Num;
              Width : in Field := 0);
procedure Get(Item : out Num; Width : in Field := 0);
```

Ha a szélesség (`Width`) 0, átugorja a helyközöket, sor és oldal vége jeleket, olvas egy előjelet, ha van (minusz jelet csak előjeles típus esetén) majd az egészek szintaxisának megfelelően veszi a számot. Ha a szélesség (`Width`) nem 0, akkor pontosan ennyi karaktert olvas be, vagy a sor vége jelig olvas (amelyik előbb jön), minden átugrott bevezető szóköz beleszámít az olvasott karakterek számába. Az `Item` paraméterbe a beolvasott karaktersorozatnak megfelelő szám kerül.
Hibajelzés:
`Data_Error` - ha nem megfelelő az olvasott jel

```
procedure Put(File  : in File_Type;
              Item  : in Num;
              Width : in Field := Default_Width;
              Base  : in Number_Base := Default_Base);

procedure Put(Item  : in Num;
              Width : in Field := Default_Width;
              Base  : in Number_Base := Default_Base);
```

Kiírja az `Item` paraméter értékét, ha az eredmény szélessége kevesebb, mint a `Width`, akkor a megfelelő számú helyközt a szám elé írja.

```
procedure Get(From : in String;
              Item : out Num;
              Last : out Positive);
```

Az adott sztring elejéről beolvas egy egész számot ugyanúgy, mint az előző `Get`, `Item`-ben a beolvasott érték, `Last`-ban az az index érték, amire `From(Last)` az utolsó beolvasott karakter lesz.
Hibajelzés:

`Data_Error` - ha nem megfelelő az olvasott jel szintaxisa, vagy ha a kapott érték nincs benne a `Num` típusban.

```
procedure Put(To : out String;
              Item : in Num;
              Base : in Number_Base := Default_Base);
```

A sztringbe kiteszi `Item` értékét, a sztring hosszát tekinti szélességértéknek.

A `Text_IO.Integer_IO` sablon csomagnak vannak nem sablon változatai az összes előre definiált egész típusra. E csomagok nevei az alaptípusból képződnek, pl.: `Ada.Integer_Text_IO`, `Ada.Long_Integer_Text_IO`.

A következő példaprogram az egész típusok input-outputját szemlélteti:

10.4. Példa *(Egész típusok inputja és outputja)*

```
with Text_Io, Sequential_Io;
use Text_Io;
procedure Int_io is

   -- a stdin-ről való olvasáshoz
   package Int_Io is new Integer_Io(Integer);
   use Int_Io;

   -- a file-ba való íráshoz, illetve olvasáshoz
   package sio is new Sequential_Io(Integer); use sio;
   F: sio.File_Type;
   I: Integer;
begin
   Create(F, Out_File, "példa.dat");
   while not End_Of_File loop
      Get(I);
      Write(F, I);
```

```
   end loop;

      -- a példa.dat fájl tartalmának kiírása
   Reset(F, In_File);
   while not End_Of_File(F)  loop
      Read(F, I);
         -- 3-as számrendszerben
      Put(I, Width => 20, Base => 3);
       -- 10-es számrendszerben
      Put(I, Width => 7);
      New_Line;
   end loop;
   Close(F);
end Int_Io;
```

A program első fele a `stdin`-ről (standard input file) egész számokat olvas, és a `példa.dat` nevű fájlba másolja. (Ha hibás adatot kap, akkor a program kivétel emelésével leáll.)

A program második fele elolvassa a `példa.dat` fájlt, az így kapott számokat kiírja a standard output fájlba, a számok 3-s számrendszerbeli alakjával együtt.

10.4.5. Valós típusok input-outputja

A `Float_IO` és a `Fixed_IO` és `Decimal_IO` beágyazott sablon csomagok definiálják a lebegőpontos és fixpontos és decimális fixpontos (tizedespontos) számok input-outputját, használat előtt példányosítani kell őket a megfelelő valós típusra.

A kezelt értékekből képzett decimális literálok lehetséges formátumai:

– Fore mező tizedespont Aft mező, ill.

– Fore mező tizedespont Aft mező E Exp mező, ahol nincs helyköz az egyes mezők között

A Fore-ban lehetnek az elején helyközök és egy előjel a negatív értékekre, az Aft csak decimális számjegyeket tartalmaz, az Exp mező előjelet és kitevőt. E mezők szélességének default értéke: lebegőpontos típusokra:

```
Default_Fore : Field := 2;
Default_Aft : Field := Num'Digits-1;
Default_Exp : Field := 3;
```

fixpontos és decimális fixpontos típusokra:

```
Default_Fore : Field := Num'Fore;
Default_Aft : Field := Num'Aft;
Default_Exp : Field := 0;
```

A csomag által szolgáltatott eljárások:

```
procedure Get(File : in File_Type; Item : out Num;
              Width : in Field := 0);
procedure Get(Item : out Num; Width : in Field := 0);
```

Ha a szélesség(`Width`) 0, átugorja a helyközöket, sor és oldal vége jeleket, olvas egy előjelet, ha van, majd a valósak szintaxisának megfelelően veszi a számot (addig olvas, amíg a beolvasott karakterek illeszkednek a valós számok szintaxisára). Ha a szélesség(`Width`) nem 0, akkor pontosan ennyi karaktert olvas be, vagy a sor vége jelig olvas (amelyik előbb jön), minden átugrott bevezető blank beleszámít a karakterek számába. Az `Item` paraméterbe a beolvasott karaktersorozatnak megfelelő szám kerül.
Hibajelzés:

`Data_Error` - ha nem megfelelő az olvasott jel szintaxisa, vagy ha a kapott érték nincs benne a `Num` típusban.

```
 procedure Put(File : in File_Type;
Item : in Num;
Fore : in Field := Default_Fore;
Aft : in Field := Default_Aft;
Exp : in Field := Default_Exp);

 procedure Put(Item : in Num;
Fore : in Field := Default_Fore;
Aft : in Field := Default_Aft;
Exp : in Field := Default_Exp);
```

Decimális literálként kiírja az `Item` paraméter értékét a `Fore`, `Aft`, `Exp` által meghatározott formátumban.

– Ha az érték negatív, vagy `Num'Signed_Zeros True` és az érték egy negatív előjelű nulla, akkor egy minusz jel is bekerül az egész részbe.

– Ha `Exp` értéke 0, akkor a kiírt szám egész része legalább annyi számjegyből áll, mint az érték egész része, (ezzel esetleg `Fore`-t figyelmen kívül hagyva), illetve az egész rész 0, ha `Item` értékének nincs egész része.

– Ha `Exp`>0, akkor a kiírt érték egész része egyetlen számjegyből áll, ami csak akkor lehet 0, ha az `Item` 0.0. Mindkét esetben, ha az egész résznek kevesebb mint `Fore` karaktere van, akkor szóközöket ír a szám elé. A törtrész számjegyeinek száma az `Aft`-ben adott, vagy 0, ha a törtrész 0. Az érték kerekített, a pontosan fél érték az utolsó helyen felfelé kerekíthető.

```
procedure Get(From : in String; Item : out Num;
              Last : out Positive);
```

Az adott sztring elejéről beolvas egy egész számot ugyanúgy, mint az előző `Get`, `Item`-ben a beolvasott érték, `Last`-ban az az index érték, amire `From(Last)` az utolsó beolvasott karakter.
Hibajelzés:

`Data_Error` - ha nem megfelelő az olvasott jel szintaxisa, vagy ha a kapott érték nincs benne a `Num` típusban.

```
procedure Put(To : out String;
      Item : in Num;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);
```

A sztringbe kiteszi `Item` értékét, olyan `Fore` értéket használ, hogy pontosan kitöltse a sztringet.

A `Text_IO.Float_IO` sablon csomagnak vannak nem sablon változatai az összes előre definiált lebegőpontos típusra. Ezen csomagok nevei például: `Ada.Float_Text_IO`, `Ada.Long_Float_Text_IO`.

A `Text_IO.Float_IO` bemutatására készült az alábbi példa, melyben egy `string`ből olvasunk egy lebegőpontos számot, majd kiírjuk három különböző formában a standard outra.

10.5. Példa *(A* `Text_IO.Float_IO` *használata)*

```
with Text_Io; use Text_Io;
procedure F_io is

   package Fio is new Float_IO (Float); use Fio;
   I: Float;
   P: Positive;

begin
   Get("  123.45678  ", I, P);
   Put(I);
   New_Line;
   Put(I, Fore=>5);
   New_Line;
   Put(I, Fore=>5, Aft=>10);
   New_Line;
end F_Io;
```

A program eredménye:

```
 1.23457E+02
    1.23457E+02
    1.2345677948E+02
```

10.4.6. Felsorolási típusok input-outputja

Az `Enumeration_IO` beágyazott sablon csomag definiálja a felsorolási típusok input-outputját. Használat előtt példányosítani kell a megfelelő felsorolási típusra. A kiíratások formátumánál a `Set` paraméterben adjuk meg, hogy nagy- vagy kisbetűket használjon, a `Width` paraméterben pedig a szélességet. Ezek default értéke:

```
Default_Width : Field := 0;
Default_Setting : Type_Set := Upper_Case;
```

Az alábbiakban áttekintjük a csomag eljárásait. Az eljárásokban szereplő `Enum` típus a sablon paraméter típusa.

```
procedure Get(File : in File_Type; Item : out Enum);
procedure Get(Item : out Enum);
```

Átugorja a szóközöket, sor és oldal vége jeleket, olvas egy, a lexikai elem szintaxisának megfelelő azonosítót (nagy- és kisbetűk ekvivalensek), vagy egy karakterliterált. Az `Item`-ben a beolvasott karaktersorozatnak megfelelő értéket adja vissza.
Hibajelzés:

`Data_Error` - ha nem megfelelő az olvasott jel szintaxisa, vagy ha a kapott érték nincs benne az `Enum` típusban.

```
procedure Put(File : in File_Type;
              Item : in Enum;
              Width : in Field := Default_Width;
              Set : in Type_Set := Default_Setting);
procedure Put(Item : in Enum;
              Width : in Field := Default_Width;
              Set : in Type_Set := Default_Setting);
```

Felsorolási literálként kiírja az `Item` paraméter értékét, a `Set` mutatja, hogy nagy- vagy kisbetűvel (karakterliterálra ennek nincs hatása). Ha az `Item`-nek megfelelő jelsorozatban `Width`-nél kevesebb karakter van, utána írja a szóközöket. Ha `Enum` karakter típus, akkor a kiírt karaktersorozat megegyezik az `Enum'Image(Item)` `Width` és `Set` által módosított értékével.

```
procedure Get(From : in String;
              Item : out Enum;
              Last : out Positive);
```

Az adott sztring elejéről beolvas egy felsorolási literált ugyanúgy, mint az előző `Get` (a sztring végét fájlvégnek tekintve); `Item`-ben a beolvasott érték, `Last`-ban pedig az az index érték lesz, amire `From(Last)` az utolsó beolvasott karakter.
Hibajelzés:

`Data_Error` - ha nem megfelelő az olvasott jel szintaxisa, vagy ha a kapott érték nincs benne az `Enum` típusban.

```
procedure Put(To : out String;
              Item : in Enum;
              Set : in Type_Set := Default_Setting);
```

A sztringbe kiírja az `Item` értékét, ugyanazon szabályok szerint, mint a fájlra vonatkozó `Put`. A sztring hosszát tekinti `Width` értéknek.

Bár a csomag specifikációja megengedi a valós típussal történő példányosítást, ez nem célja ennek a csomagnak, ezért ebben az esetben a nyelv nem definiálja a működést.

Lássunk egy példát az `Enumeration_IO` használatára:

10.6. Példa *(A* `Enumeration_IO` *használata)*

```
with Text_Io; use Text_Io;
procedure Enu_Io is
   type Gyümölcs is (Alma, Narancs, Barack);
   package Gyümölcs_Io is new Enumeration_Io (Gyümölcs);
   use Gyümölcs_Io;

   G: Gyümölcs;
   P: Positive;
begin
     Get(" Alma másik alma harmadik alma", G, P);
     Put(G);
     New_Line;
     Put(G, Width=>10, Set=>Lower_Case);
end Enu_Io;
```

A programban szerepel egy `Gyümölcs` nevű felsorolás típus, amelynek elemei: `Alma, Narancs, Barack`

A program futása során a „`Alma másik alma harmadik alma`" sztringből kiolvassa az első elemet – jelen esetben ez pont egy `Gyümölcs` típusú érték – majd a kiolvasott `Gyümölcs` típusú értéket kiírja a standard output fájlba.

A program eredménye:

```
ALMA
alma
```

10.5. Wide_Character és Wide_String input-output

Wide_Character és Wide_String értékek szöveges input-outputjának megvalósítása a Wide_Text_IO csomagban található. A csomag specifikációja megegyezik a Text_IO csomagéval, persze azzal a módosítással, hogy a Get, Look_Ahead, Get_Immediate, Get_Line, Put és Put_Line műveletekben a Character és String típusokat Wide_Character, illetve Wide_String típusokra cseréljük.

A Text_IO-hoz hasonlóan a beágyazott sablon numerikus csomagoknak létezik nem sablon változata is az előredefiniált numerikus típusokra, és ezeknek névképzése is az ottanival analóg.

10.6. Adatfolyam input-output

A fájlok adatfolyam-orientált (stream-orientált) kezelését az Ada nyelv a Streams.Stream_IO és Text_IO.Text_Streams csomagokkal támogatja.

10.6.1. A Streams.Stream_IO csomag

Az adatfolyam fájlok kezelését a nyelv a Streams. Stream_IO csomag eljárásain keresztül valósítja meg. Az ilyen fájlok kezelhetők szekvenciálisan a Read és Write műveletekkel, vagy pozicionálisan (mármint ha az implementáció támogatja), egy relatív index megadásával. A csomag deklarációja az alábbi:

```
with Ada.IO_Exceptions;
package Ada.Streams.Stream_IO is

   type Stream_Access is access all Root_Stream_Type'Class;

   type File_Type is limited private;

   type File_Mode is (In_File, Out_File, Append_File);

   type  Count  is range 0 .. implementation-defined;
   subtype Positive_Count is Count range 1 .. Count'Last;

   procedure Create (File : in out File_Type;
                     Mode : in File_Mode := Out_File;
                     Name : in String   := "";
                     Form : in String   := "");
```

```
procedure Open (File : in out File_Type;
                Mode : in File_Mode;
                Name : in String;
                Form : in String := "");

procedure Close  (File : in out File_Type);
procedure Delete (File : in out File_Type);
procedure Reset  (File : in out File_Type;
                  Mode : in File_Mode);
procedure Reset  (File : in out File_Type);

function Mode (File : in File_Type) return File_Mode;
function Name (File : in File_Type) return String;
function Form (File : in File_Type) return String;

function Is_Open (File : in File_Type) return Boolean;
function End_Of_File (File : in File_Type) return Boolean;
```

A `Create`, `Open`, `Close`, `Delete`, `Reset`, `Mode`, `Name`, `Form`, `Is_Open`, és `End_of_File` műveletek hatása ugyanaz, mint a `Sequential_IO` csomag megfelelő műveleteié.

```
function Stream (File : in File_Type) return Stream_Access;
```

A `Stream` függvény egy `Stream_Access` mutatót ad vissza a paraméter `File_Type` objektum alapján, ezzel lehetővé teszi, hogy a `Read` és `Write` műveletek segítségével különböző típusú értékeket olvassunk ugyanabból a fájlból.

```
procedure Read (File : in  File_Type;
      Item : out Stream_Element_Array;
      Last : out Stream_Element_Offset;
      From : in  Positive_Count);

procedure Read (File : in  File_Type;
      Item : out Stream_Element_Array;
      Last : out Stream_Element_Offset);

procedure Write (File : in File_Type;
     Item : in Stream_Element_Array;
     To : in Positive_Count);

procedure Write (File : in File_Type;
     Item : in Stream_Element_Array);
```

A `Read` és `Write` műveletek ekvivalensek a megfelelő `Streams` csomagbeli műveletekkel. A `Read` műveletek `Mode_Error` kivételt váltanak ki, ha a fájl nem `In_File` üzemmódú. A `Write` műveletek viszont akkor váltanak ki `Mode_Error` kivételt, ha a fájl `In_File` módú. A `Positive_Count` utolsó paraméterrel rendelkező `Read` és `Write` műveletek a megadott indexen kezdenek el olvasni, illetve írni.

```
procedure Set_Index(File : in File_Type;
                    To : in Positive_Count);

function Index(File : in File_Type) return Positive_Count;
```

Az `Index` függvény az aktuális fájlpozíciót adja vissza `Stream_Element`-ben számolva. A fájl első elemének indexe 1. A `Set_Index` művelet segítségével ez az aktuális pozíció változtatható meg.

Ha az adott fájl nem pozicionális, akkor mindkét művelet – sőt a pozicionális `Read` és `Write` műveletek is – `Use_Error` kivételt váltanak ki.

```
function Size (File : in File_Type) return Count;
```

A `Size` függvény a fájl aktuális méretét adja vissza `Stream_Element`-ben számolva.

```
procedure Set_Mode(File : in out File_Type;
                   Mode : in File_Mode);
```

A `Set_Mode` művelet segítségével megváltoztathatjuk a fájl üzemmódját. Ha az új üzemmód `Append_File`, akkor a fájlt a végére pozicionálja, egyébként a fájlpozíció nem változik.

```
procedure Flush(File : in out File_Type);
```

A `Flush` művelet szinkronizálja a külső fájlt a belső fájlobjektummal. `Mode_Error` kivételt vált ki, ha a fájl `In_File` üzemmódú.

```
Status_Error : exception
  renames IO_Exceptions.Status_Error;
Mode_Error : exception renames IO_Exceptions.Mode_Error;
Name_Error : exception renames IO_Exceptions.Name_Error;
Use_Error  : exception renames IO_Exceptions.Use_Error;
Device_Error : exception
  renames IO_Exceptions.Device_Error;
End_Error  : exception renames IO_Exceptions.End_Error;
Data_Error : exception renames IO_Exceptions.Data_Error;
```

```
private
  ...
end Ada.Streams.Stream_IO;
```

A Streams.Stream_IO használatára mutatunk egy primitív példát. A program egy előre megadott nevű fájl (forrás) maximum első 100000 bájtját egy másik fájlba másolja (másolat).

10.7. Példa *(*Streams.Stream_IO *használata)*

```
procedure Stream_Pl is
   F,G: File_Type;

   Adat_Buffer: Stream_Element_Array(1..100000);
   Számláló: Stream_Element_Offset ;
begin
   Open(F, In_File, "forrás");
   Read(F, Adat_Buffer, Számláló);
   Close(F);

   Create(G, Out_File, "másolat");
   Write(G, Adat_Buffer(1..Számláló) );
   Close(G);

   Text_Io.Put(Stream_Element_Offset'Image(Számláló));
   Text_Io.Put(" db byte-ot sikerült átmásolni");
   Text_Io.Put_Line;

end Stream_Pl;
```

10.7. A Text_IO.Text_Streams csomag

A Text_IO.Text_Streams csomagon keresztül a text fájlokat adatfolyam fájlokként kezelhetjük. A csomag deklarációja az alábbi:

```
with Ada.Streams;
package Ada.Text_IO.Text_Streams is
 type Stream_Access is access all
    Streams.Root_Stream_Type'Class;

 function Stream (File : in File_Type) return Stream_Access;
end Ada.Text_IO.Text_Streams;
```

A Stream függvény hatása pontosan megegyezik a Streams.Stream_IO csomag Stream műveletével. Az egy textfájlhoz kapcsolódó adatfolyamon

elvégzett műveletek nem befolyásolják a textfájl aktuális oszlop-, sor- és lapsorszámait.

10.8. A Wide_Text_IO.Text_Streams csomag

```
with Ada.Streams;
package Ada.Wide_Text_IO.Text_Streams is
 type Stream_Access is access all
    Streams.Root_Stream_Type'Class;

 function Stream (File : in File_Type) return Stream_Access;
end Ada.Wide_Text_IO.Text_Streams;
```

A fenti csomag Wide_Text változata, ugyanarra a célra használható, csak Wide_Text_IO-beli fájlokra.

10.9. Az input-output során fellépő kivételek

Azokat a kivételeket, amelyek az input-output műveletek során előfordulhatnak, az IO_Exceptions csomag definiálja:

```
package Ada.IO_Exceptions is
 pragma Pure(IO_Exceptions);

 Status_Error : exception;
 Mode_Error : exception;
 Name_Error : exception;
 Use_Error  : exception;
 Device_Error : exception;
 End_Error  : exception;
 Data_Error : exception;
 Layout_Error : exception;

end Ada.IO_Exceptions;
```

Status_Error hibajelzést akkor kaphatunk, ha olyan fájlra akarunk műveletet elvégezni, amely nincs nyitva, vagy olyat akarunk megnyitni, amelyet már megnyitottunk.

Mode_Error hibaüzenetet az alábbi esetekben kaphatunk:

- ha olyan fájlból akarunk olvasni, vagy olyan fájlra akarunk fájlvége tesztet végezni amelynek üzemmódja Out_File vagy Append_File,

- ha olyan fájlba akarunk írni, amely In_File módra lett megnyitva,
- a Text_IO csomag Set_Input, Skip_Line, End_Of_Line, Skip_Page, és End_Of_Page műveletei Out_File vagy Append_File módú fájlok esetén,
- a Text_IO csomag Set_Output, Set_Line_Length, Set_Page_Length, Line_Length, Page_Length, New_Line vagy New_Page műveletei ha a fájl In_File módra van megnyitva.

Name_Error kivételt váltanak ki a Create és Open műveletek, ha a névnek megadott sztring paraméter nem azonosít külső fájlt (pl. a fájlrendszernek nem megfelelő, vagy több fájlt is azonosít).

Use_Error hibajelzést kapunk azokban az esetekben, ha olyan műveletet akarunk elvégezni, amelyet az adott külső fájlra nem lehet.

Device_Error hibajelzést kapunk azokban az esetekben, amikor valamilyen rendszerhibából fakadóan az adott műveletet nem lehet elvégezni.

End_Error kivételt vált ki, ha túl akarunk lépni vagy olvasni a fájl végén.

Data_Error kivételt vált ki a Read eljárás, ha a beolvasott érték nem értelmezhető a kívánt (al)típus értékének. Ugyanezt okozhatja az is, ha a Text_IO csomag Get művelete során a beolvasott érték nem felel meg a megkívánt szintaxisnak, vagy nem esik bele a kívánt (al)típus értékkészletébe.

Layout_Error hibajelzést kapunk a szöveges input-output során, ha a Col, Line és Page értékek valamelyike túllépi a Count'Last értéket. Akkor is ezt a hibajelzést kapjuk, ha olyan értékre akarjuk beállítani az aktuális oszlop vagy sor értékeket, amelyek nagyobbak a maximális sor- vagy oldalhosszak (természetesen a nemkorlátos esetek kivételével). Továbbá a Put művelet is ezt a kivételt váltja ki, ha egy sztringbe túl sok karaktert akarunk kiírni.

Az egyes implementációk dokumentálják, hogy pontosan milyen feltételek között váltódnak ki a Name_Error, Use_Error and Device_Error kivételek.

10.10. Fájlok megosztása

A nyelv nem specifikálja, hogy egy külső fájl hozzárendelhető-e egyidejűleg több fájl objektumhoz is. Ha az adott implementáció támogatja az ilyen osztott hozzáférést, akkor az alábbi szabályoknak kell eleget tennie:

- egy szövegfájl objektumon elvégzett műveletek nem lehetnek hatással más fájlobjektumok aktuális oszlop, sor és oldalszámaira,
- A Standard_Input és Standard_Output fájloknak különböző külső fájlokhoz kell tartozniuk, így az egyik fájl műveletei nem lehetnek hatással a másik fájlra,

- Közvetlen elérésű és adatfolyam fájlok esetén az aktuális index a fájl objektum sajátja, így egy fájl objektumra elvégzett műveletnek nem lehet hatása másik objektum aktuális indexére,
- Közvetlen elérésű és adatfolyam fájlok esetén az aktuális fájlméret a külső fájl jellemzője.

10.11. Az SVGALib könyvtár

A Linux operációs rendszereken nagyon elterjedt a GNAT Ada fordító. A GNAT-hoz léteznek olyan kiegészítő csomagok, melyekben sok hasznos függvény, illetve eljárás van implementálva.

Az egyik ilyen csomag (csomag gyűjtemény) a SVGALib, mely a Linux operációs rendszer alatt grafikus felületű programok írásához biztosít eljárásokat.

A csomag eljárásai és függvényei lényegében a GNU Linux vgalib könyvtárában szereplő C függvényeket hívják meg. A csomag bizonyos korlátozások mellett szabadon felhasználható és az Internetről szabadon letölthető.

A SVGALib könyvtár Ada95-tel kapcsolatot teremtő csomagokat Darren J. Moffat (moffatd@dcs.gla.ac.uk) írta. Ebben a fejezetben e csomagot mutatjuk be a teljesség igénye nélkül.

10.11.1. Az svgalib.ads

Az svgalib.ads a grafikus alkalmazásokhoz szükséges alapvető műveleteket és típusokat tartalmazza.

```
Invalid_Vga_Mode : exception;
```

Kivétel, mely helytelenül beállított vga mód esetén váltódik ki.

```
type VgaMode is ...
```

A VgaMode egy felsorolási típus, mely a különböző vga módok azonosítására szolgál.

```
type Point is record
   X, Y : Integer;
end record;
```

A Point segítségével egy pont koordinátáit tárolhatjuk.

```
subtype RGBNumber is Integer range 0..255;
type RGBColour is record
   Red : RGBNumber;
   Green : RGBNumber;
   Blue : RGBNumber;
end record;
```

Az `RGBColour` egy szín RGB (Vörös, Zöld, Kék) alakú tárolására használatos.

```
procedure Set_Mode (Mode : VgaMode := G320x200x16);
```

A `Set_Mode` beállítja a kívánt grafikus üzemmódot.

```
function Has_Mode (Mode : VgaMode := G320x200x16)
   return Boolean;
```

Ellenőrzi, hogy a megadott grafikus üzemmódot támogatja-e az aktuális rendszer.

```
function Set_Flip_Char (C : Integer) return Boolean;
```

A grafikus és a szöveges üzemmódok közti váltásoknál használt karaktert változtatja meg.

```
procedure Clear;
```

Törli a képernyőt.

```
procedure Flip;
```

A grafikus és a szöveges üzemmódok között vált, azok tartalmának változtatása nélkül, így olyan programok is írhatók, melyek egyaránt használnak mind grafikus, mind szöveges üzemmódot.

```
function Get_Xdim return Integer;
function Get_Ydim return Integer;
function Get_Colours return Natural;
```

Az aktuális üzemmódról adnak infomációt (szélesség, magasság, színek száma).

```
procedure Set_Palette (Palette : in Natural;
                       Colour : in RGBColour);
procedure Get_Palette (Palette : in Natural;
                       Colour : out RGBColour);
```

A paletta tartalmát lehet-e eljárásokkal módosítani, lekérdezni.

```
procedure Screen_Off;
procedure Screen_On;
```

Ki-, illetve bekapcsolja a képernyőfrissítést. Bizonyos videóvezerlőknél sebességnövekedést eredményez, ha az adatok videómemóriába való írásakor használjuk.

```
procedure Set_Colour (Colour : in Integer);
procedure Set_Color (Colour : in Integer)
   renames Set_Colour;
```

Az aktuális szín beállítása.

```
procedure Draw_Pixel (P : in Point);
procedure Draw_Point (P : in Point) renames Draw_Pixel;
```

Pontot rajzol ki az aktuális színnel.

```
procedure Draw_Line (T,B : in Point);
```

Vonalat rajzol ki az aktuális színnel.

10.11.2. Az svgalibmouse.ads

A `svgalibmouse.ads` a grafikus alkalmazások esetén az egérkezeléssel kapcsolatos műveleteket és típusokat tartalmazza.

```
Mouse_Error : exception;
```

Az egérkezelés során előforduló hibák jelzésére készített kivétel.

```
type MouseType is (Microsoft, MouseSystems, MMSeries,
   Logitech, BusMouse, PS2);
```

Az egértípusok beállításához használt felsorolási típus.

```
type MouseButton is (Right, Middle, Dummy, Left);
```

Az egérgombok azonosítására használt típus.

```
procedure Mouse_Init (MT : MouseType := Microsoft;
          Device : String := "/dev/mouse";
          Rate : Integer := Default_Sample_Rate);

function Mouse_Close return Boolean;
```

Egér inicializálása és az egérműveletek befejezése.

```
procedure Set_Default_Event_Handler;
function Mouse_Update return Boolean;
procedure Mouse_Wait_For_Update;
```

Az egéresemények kezelése.

```
procedure Mouse_Set_Position (P : Point);
function Mouse_Get_Position return Point;
```

Az egér helyzetének beállítása, lekérdezése.

```
function Mouse_Get_Button return MouseButton;
```

A lenyomott gomb lekérdezése.

10.12. Feladatok

1. Készítsünk programot, mely a képernyőn (szöveges módban) kirajzol egy kört!
2. Készítsünk programot, mely egy szöveges fájlt egy másikba másol soronként, kihagyva a #-el kezdődő sorokat!
3. Készítsünk programot, mely megszámolja egy szöveges fájl sorait, szavait és karaktereit!
4. Készítsünk programot, melynek bemenete két, egész számokat tartalmazó rendezett fájl, kimenete pedig a két bemenő fájl összefésüléseként kapott rendezett fájl! (Valósítsuk meg a programot mind szöveges, mind bináris fájlok esetén!)

11. Fejezet

Sablonok

Lényeges, hogy a könyvtári egységeket elég általánosan meg lehessen írni, és azokat széles körben, sok feladatra alkalmazni lehessen. Bizonyos általánosság elérhető az alprogramokkal: ezeket paramétereik jó megválasztásával sok hasonló feladat megoldására felhasználhatjuk. Az alprogramok paraméterei azonban csak objektumok – változók, kifejezések – lehetnek. A könyvtári egységek újrafelhasználhatóságát, többalakúságát (polimorfizmus) lényegesen sokoldalúbban támogatják a *sablonok*. A sablon programegység fogalma lehetővé teszi, hogy alprogramokat és csomagokat típusokkal és alprogramokkal is paraméterezzünk, és ezzel a programegységek általánossága tovább növelhető.[1]

Egy sablont nem lehet közvetlenül végrehajtani. A sablon egy minta, ami alapján végrehajtható egyedek, úgynevezett *példányok* hozhatók létre. Egy példány létrehozását *példányosításnak* nevezzük. A példányosítás mindig *fordítási időben* hajtódik végre.

Egy sablon definiálása magában foglalja a sablon formális paramétereinek megadását, a sablon (csomag vagy alprogram) specifikációját, valamint a sablon törzsét. Egy példányt a sablon formális paramétereinek aktuális paraméterekkel történő megfeleltetésével lehet létrehozni. Az eredmény egy olyan programegység (csomag vagy alprogram, ami a sablon volt), amit úgy kapnánk, hogy a sablon formális paramétereit a sablonban kicseréljük a megadott aktuális paraméterekre.[2]

Természetesen egy sablonból – több példányosítással – *több példányt* is létrehozhatunk. Minden példány egy újabb, immár végrehajtható programegység lesz: alprogram sablon esetén a példány egy meghívható alprogram, sablon csomag esetén egy csomag, ami egyedeket (objektumokat, alprogramokat, stb.) tartalmazhat.

[1] Az Ada95-ben megjelentek az alprogramra mutató típusok. Ezek segítségével is lehet alprogramokat paraméterként átadni. (Lásd 3.6.)

[2] A sablonokat az – egyes programozási nyelvekben, például a C-ben használatos – makrókkal lehet rokonságba hozni.

11.1. Sablonok definiálása

A sablonok lehetnek *alprogram- (eljárás- és függvény-) sablonok* valamint *sablon csomagok*. A sablonokat ugyanúgy írjuk, mint az egyszerű alprogramokat és csomagokat, csak a specifikáció elé kerül a `generic` alapszó és a sablon formális paramétereinek felsorolása. Egy sablon paraméterei lehetnek *objektumok, típusok, alprogramok és csomagok*. Ebben a példában egy típus és egy alprogram a paraméter. Az alprogram paramétert a `with` kulcsszó vezeti be:

11.1. Példa

```
generic
  type Szám is private;
  with function Mývelet( A, B: Szám ) return Szám;
function Iterált( A: Szám; N: Positive ) return Szám;

function Iterált( A: Szám; N: Positive ) return Szám is
  X: Szám := A;
begin
  for I in 2..N loop
    X := Mývelet(X,A);
  end loop;
  return X;
end Iterált;
```

A sablon alprogramokhoz mindig tartozik deklaráció is, a törzs ezt nem helyettesítheti. Az alábbi kódrészlet például helytelen:

```
generic
  type Szám is private;
  with function Mývelet( A, B: Szám ) return Szám;
function Iterált( A: Szám; N: Natural ) return Szám is
  X: Szám := A;
begin
  ...
```

Mint a fenti példa mutatja, a sablonokat használhatjuk algoritmus absztrakciók megfogalmazására. Egy másik gyakori alkalmazás az absztrakt adattípusok definiálása. Legyen most a paraméter egy típus és egy objektum:

```
generic
  type Elem is private;
  Max_Méret: in Positive;
package Vermek is
  type Verem is limited private;
  procedure Push ( V: in out Verem; E: in Elem );
```

```
  ...
private
  ...
end Vermek;

package body Vermek is
  ...
end Vermek;
```

Hasonlóan az alprogramok `in` módú paramétereihez, a sablon formális paraméterei között is lehetnek olyanok, amelyek *feltételezett értékkel* rendelkeznek. Ez megkönnyítheti a sablon felhasználását, hiszen az ilyen formális paramétereknek nem kötelező a példányosítás során aktuálist megfeleltetni:

```
generic
  type Elem is private;
  Kapacitás: in Positive := 100;  -- a feltételezett érték 100
package Sorozatok is
  ...
end Sorozatok;
```

Egy sablon formális paraméterlistáján belülre írhatunk `use` záradékot is. Ennek előnyeit egy későbbi példában tapasztaljuk majd meg. (Lásd 252. o.)

A sablon programegységeket mindenütt elhelyezhetjük, ahol egyszerű programegység állhat. Szerepelhetnek tetszőleges deklarációs részben, például alprogramében vagy blokk utasításában. Csomag specifikációjába azonban csak sablon deklaráció kerülhet, a sablon törzsének a megadása a csomag törzsében, a törzs deklarációs részében kell, hogy álljon.

```
package P is
  generic procedure A;  -- paraméter nélküli eljárássablon
end P;
package body P is
  procedure A is begin null; end;
end P;
```

Sablonok deklarációja és sablonok törzse *fordítási egység* is lehet. Ne feledjük el, hogy minden ilyen esetben a sablon deklarációja és a sablon törzse két külön fordítási egység. Mint azt az előzőekben már említettük, minden sablonhoz (az alprogram sablonokhoz is) kell, hogy írjunk deklarációt, ezért, ha egy alprogram sablont *könyvtári egységként* szeretnénk megírni, mindig két fordítási egységet kell készítenünk.[3]

[3]Sablon csomagoknál van egy kis kibúvó ez alól a szabály alól: ha egy csomag specifikációja nem tartalmaz olyan deklarációt, amit a csomag törzsében ki kell egészíteni, a csomaghoz nem kötelező törzset készíteni. (Lásd 7.2.)

A sablon könyvtári egységre ugyanúgy `with` záradékkal kell hivatkozni, mint más könyvtári egységre. A sablon csomagokra a `use` záradék nem használható, hiszen ennek nincs is értelme: a sablon komponensei csak a sablon példányosításával, a példányban válnak „valóságos" dolgokká, ezért a sablonok komponenseire nem, csak a példányok komponenseire lehet hivatkozni (és a minősített hivatkozást a `use` záradékkal feloldani). A `use` záradékot tehát egy példányra már kiadhatjuk. (Lásd 232. o.)

```
with Vermek;      -- egy sablon csomag könyvtári egység
-- use Vermek;    értelmetlen, hibás lenne
procedure Kifejezés_Kiértékelés is ...
```

A sablonon belül a sablon neve az *aktuális példányt* jelöli, nem a sablont. Ily módon írhatunk rekurzív alprogram sablonokat is. A fentebb bemutatott `Iterált` függvénysablon törzsét például így is írhatjuk:

11.2. Példa

```
function Iterált( A: Szám; N: Positive ) return Szám is
begin
  if N = 1 then return A;
  else return Mývelet(Iterált(A,N-1),A);
  end if;
end Iterált;
```

Egy fontos megkötés, hogy alprogramsablon neve nem lehet operátor. Nem írhatunk például `"+"` nevű függvénysablont.

A sablonok talán legfőbb erőssége a típussal való paraméterezhetőség. Ennek támogatására a sablon formális paraméter típusok megadásának egy széles választékát kínálja a nyelv. A formális paraméter típus megszorítást tehet az aktuális paraméter típus osztályára is (diszkrét, tömb, mutató, jelölt típus, stb.), ezzel implicit módon típusműveleteket deklarálva hozzá, amiket például a sablon törzsének definiálásakor használhatunk:

11.3. Példa

```
generic
  type Elem is private;
  type Index is (<>);
  type Vektor is array (Index range <>) of Elem;
  with function "<" ( A, B: Elem ) return Elem;
function Maximum_Hely ( V: Vektor ) return Elem;

function Maximum_Hely ( V: Vektor ) return Elem is
  M: Index := V'First;
begin
  for I in V'Range loop
    if V(M) < V(I) then M := I; end if;
```

```
  end loop;
  return M;
end Maximum_Hely;
```

A példányosítás során az `Index` paraméternek valamilyen diszkrét típus feleltethető meg, a `Vektor`-nak pedig olyan tömb, amely az `Elem` és `Index` formálisaknak megfeleltetett aktuálisakból a leírt módon képzett. A függvény törzsében használhatjuk a tömb típusosztály műveleteit, például az indexelést és a `'First` és `'Range` attribútumokat.

A sablonok típus és egyéb formális paramétereiről – és például a paraméterek feltételezett értékéről – a későbbiekben (lásd 11.3.) részletesebben is szót ejtünk.

11.2. Sablonok példányosítása

Egy sablonból úgy hozunk létre egy példányt, hogy a sablon formális paramétereinek aktuális paramétereket feleltetünk meg. Tekintsük az `Iterált` függvénysablon néhány lehetséges példányosítását. Mint látható, példánnyal felüldefiniálhatunk operátorokat is:

```
function Szorzás is new Iterált(Positive,"+");
function Hatványozás is new Iterált(Positive,Szorzás);
function "**" is new Iterált(Positive,"*");
```

A sablonok nem végrehajtható programegységek, csak példányok létrehozására használhatók. A példányok ezzel szemben végrehajtható programegységek, de belőlük újabb példány már nem hozható létre. Az előbb létrehozott `Hatványozás` és `"**"` függvényeket így használhatjuk:

```
if Hatványozás(3,7) /= 3**7 then Put_Line("Na ne..."); end if;
```

A *sablonparaméterek megfeleltetésénél*, csakúgy, mint az alprogramok esetén (lásd 5.1.4.), a *pozícionális* és a *névvel jelölt formát* is alkalmazhatjuk. A névvel jelölt formánál a paraméter-megfeleltetések sorrendje tetszőleges. Ha a két formát keverjük, a pozícionális paraméterek meg kell, hogy előzzék a névvel jelölteket.

```
package Int_Vermek is new Vermek(Integer,Max_Méret=>100);
```

A feltételezett értékkel rendelkező formális paramétereknek nem kötelező a példányosításkor aktuálist megfeleltetni:

```
package Int_Sorozatok is new Sorozatok(Integer);
```

Egy sablon példányosítása hatásában megfelel annak, mint a példány deklarációja és törzsének megadása a példányosítás helyén: onnantól kezdve a programszövegben már hivatkozhatunk a példányra. Példányosítás írható tetszőleges deklarációs részben, még csomag specifikációs részében is.[4]

Az előző szakaszban beszéltünk arról, hogy a sablon csomagok komponenseire (a sablonon kívül) nem lehet hivatkozni. Más a helyzet a példánnyal. A példány az előzőek alapján egy közönséges csomag:

```
package Int_Vermek is new Vermek(Integer,Max_Méret=>100);
V: Int_Vermek.Verem;
...
Int_Vermek.Push(V,42);
...
```

A példányokra a minősített hivatkozás feloldása is lehetséges, a `use` záradékot használva. A `use` záradék – a `with` záradékkal ellentétben – nem csak fordítási egység előtt szerepelhet, hanem egy fordítási egységen belül is (lásd 8.3.), gyakran ebben a formában:

```
package Int_Vermek is new Vermek(Integer,Max_Méret=>100);
use Int_Vermek;
V: Verem;
...
Push(V,42);
...
```

Egy sablon programegység példányosítása lehet *fordítási egység* is:

```
with Text_IO; use Text_IO;
package Nat_IO is new Integer_IO(Natural);
```

11.2.1. A sablon-szerződés modell

Egy sablon példányosításakor az aktuális és formális sablon paraméterek megfeleltetése szigorú szabályoknak kell, hogy eleget tegyen. Minden aktuális paraméter meg kell, hogy feleljen a formális paraméter által támasztott követelményeknek. Hogy mik ezek a követelmények? Elsősorban a formális paraméter fajtája (paraméter objektum, típus, alprogram vagy csomag): egy formális paraméter objektumnak például nem feleltethetünk meg egy típust vagy egy alprogramot. Egy paraméter objektumnál további megkötéseket határoz meg a formális paraméter típusa, egy paraméter típusnál a típusosztálya vagy egy paraméter alprogramnál az alprogram paramétereinek száma

[4]Ez egy kis különbséget jelent az (a.) példányosítás és (b.) a példány deklarációja és törzsének megadása között: törzs – és így (b.) – ugyanis nem állhat csomag specifikációs részében, míg példányosítás igen.

és típusa. (További megkötéseket ismerünk majd meg a sablon paraméterek részletes bemutatásánál. Lásd: 11.3.) Az aktuális és formális paraméterek egyeztetésének módját a sablon-szerződés modell segítségével szemléltetjük. Ebben a szakaszban ezt a modellt tárgyaljuk.

A sablon programegységekben szereplő dolgok (deklarációk és törzsek) értelmezése, érvényességének[5] ellenőrzése *a sablon fordításakor* történik: a példányba kerülő dolgok, szerkezetek jelentése a sablonbeli megfelelőik jelentésének másolata. Ez teszi igazán biztonságossá a sablonok használatát: a sablonban előforduló hibák nem akkor derülnek ki, amikor a sablont példányosítjuk és a példányt lefordítjuk, hanem már a sablon lefordításakor.

A sablon deklarációja egy szerződést fogalmaz meg a sablon törzse és a példányosítás között. A törzs érvényessége nem függ a példányosítás érvényességétől, és fordítva, a példányosítás érvényessége sem függ a törzs érvényességétől. Ha mind a kettő megfelel a sablon deklarációnak, azaz betartja a deklaráció által előírt szerződést, akkor a példány is érvényes lesz.[6]

A szerződés szövege a sablon formális paramétereinek specifikációja. Mutattunk már – lásd 230. o. – és még mutatni is fogunk – lásd 11.3.2. – arra példát, hogy a sablon formális paraméter típus típusosztálya meghatároz egy, minden lehetséges aktuális paraméter típus esetén használható típusművelet-halmazt. A szerződés pontjainak lerögzítésére egyéb lehetőségeket ismerünk majd meg a sablon formális paraméterek fajtáinak részletes tárgyalásánál. (Lásd 11.3.)

Hogyan ellenőrzi a fordító, hogy a szerződés betartatik? Először is megvizsgálja, hogy a specifikáció önmagában érvényes-e. Ehhez a „lehető legjobbakat” feltételezi a példányosításról, és ezt a példányosítás fordításakor, a példányosítással való összevetésnél ellenőrzi is. A másik oldalról, a törzzsel való egyeztetésnél viszont a „lehető legrosszabbat” tételezi fel a példányosításról: ez eredményezi azt, hogy bármi is legyen a példányosítás, ami megfelel a sablon deklarációnak, az nem kerül összeütközésbe a sablon törzsével sem.[7]

A szerződés modellnek köszönhetően a sablon törzsének és példányosításának érvényessége egymástól külön vizsgálható. Ez egyrészről azt eredményezi, hogy nem kell minden példányosítás érvényességét újra ellenőrizni, ha a sablon törzsét módosítjuk, ezáltal a programok karbantarthatósága, módosíthatósága növekszik. Másrészről, a sablon törzsét sem kell egy újabb példányosítás elkészítésekor összevetni a példányosítással. A sablon helyes használatához nincs is szükség a sablon törzsének ismeretére. Ez a moduláris és a csoportmunkában végzett programfejlesztést segíti.

[5] Azt mondjuk, hogy egy program érvényes, ha a fordító nem jelez benne hibát.

[6] A nyelv szabályainak egy jól meghatározott részhalmazára (a szerkesztés közbeni és a futási idejű ellenőrzésekre) nem érvényesek a szerződés modell kikötései.

[7] Emiatt a szabály miatt, ha egy deklaráció egy sablon csomag törzsében nem érvényes, segíthet, ha a deklarációt átmozgatjuk a sablon csomag specifikáció privát részébe.

11.3. Sablon paraméterek

A sablonok formális paraméterei négyfélék lehetnek: objektumok, típusok, alprogramok és csomagok. Az alábbiakban részletezni fogjuk a négy paraméterfajta által nyújtott lehetőségeket. Kitérünk a formális paraméterek specifikációjának szintaxisára, a paraméterek feltételezett értékére és az aktuális paraméterekkel való megfeleltetés szabályaira, a szerződés modellre is.

11.3.1. Paraméter objektumok

Egy sablonnak adhatunk át paraméterül *értékeket* és *változókat*. A két lehetőséget a *paraméter* objektum *módja* különbözteti meg egymástól:

```
generic
  E: in Integer;
  V: in out Integer;
```

Az `in` módú paraméter a sablon programegység egy konstansa lesz: a konstans értékét az aktuális paraméter példányosításkori értéke határozza meg. Ezzel szemben az `in out` módú paraméter a sablonon belül az aktuális paraméterként megadott változó egy átnevezése lesz. (Az `in out` módú formális paraméternek csak változót lehet aktuálisként megfeleltetni, míg az `in` módúnak bármilyen kifejezést.) Úgy is fogalmazhatunk, hogy az `in` módú paraméterek másolat szerint adódnak át, míg az `in out` módúak hivatkozás szerint.

11.4. Példa

```
with Text_IO; use Text_IO;
procedure Obj_P_Mód is
  generic
    E: in Integer;
    V: in out Integer;
  procedure G;
  procedure G is
  begin
    V := E*V; -- E := E*V; hibás, E nem lehet baloldalt
  end G;
  X: Integer := 6;
  procedure P is new G(X,X);  -- ua., mint G(6,X)
begin
  X := X+1;
  P;
  Put_Line(Integer'Image(X));  -- az X értéke 42
end;
```

A paraméter objektumok alapértelmezés szerint `in` módúak:

```
generic
  A: Integer;  -- in módú
```

Az `in` módú formális paraméter objektumok rendelkezhetnek *feltételezett értékkel.* Ebben az esetben a példányosításkor nem szükséges ehhez a formális paraméterhez aktuálist kapcsolni:

```
generic
  X: in Integer := 42;
package G is end G;
package P is new G;
```

11.3.2. A paraméter típusok fajtái

A sablon formális paraméter típusok nagyon sokfélék lehetnek. Segítségükkel – a sablon szerződés modellel összhangban – sokféle megkötést tehetünk az aktuális paraméterre. Ebben a szakaszban ezeket a lehetőségeket fogjuk sorra venni. Kezdjük mindjárt a legegyszerűbbel. Ha semmilyen különleges kikötést nem teszünk az aktuális paraméterre, a formálist így adhatjuk meg:

```
generic
  type Elem is private; -- "átlagos" formális paraméter típus
package Listák is
...
```

Erre a fajtára már a fejezet elején is láttunk példákat, gondoljunk a `Verem` típust megvalósító sablon csomagra vagy az `Iterált` függvénysablonra.

Gyakran alprogramokat is megadunk paraméterként emellett a formális paraméter típus mellett. Ezek a paraméter alprogramok a típus azon típusműveletei lehetnek, amiket használni szeretnénk a sablonon belül. Itt ismét az `Iterált` függvénysablont hozhatjuk fel példának, vagy az alábbi, leegyszerűsített elemenkénti feldolgozás sémát:

11.5. Példa

```
generic
  type Elem is private;
  with function "<" ( A, B: Elem ) return Boolean;
  with procedure Beolvas ( SX: in out Boolean; DX: out Elem;
                           X: in File_Type );
  with procedure Kiír ( X: in File_Type; E: in Elem );
procedure Összefésül( X, Y: in File_Type; Z: in File_Type );
```

Ada programokban kétféle helyen találkozhatunk `private` jelzésű típussal. Az egyik hely egy átlátszatlan típus definiálása egy csomagspecifikáció látható részében, a másik pedig a most megismert sablon formális

paraméterlista. Érdekes összevetni a `private` kulcsszó szerepét a két esetben. Az első esetben a `private` azt jelenti, hogy a típus reprezentációja el van rejtve a *külvilág elől* (a csomagspecifikáció elzárt, privát részében), ezért az nem látható a csomagon kívül. A típusértékek manipulálására a csomagot használó programegységekben csak a típus látható műveletein keresztül van lehetőség: ezek a látható típusműveletek általában azok, amelyek a csomagspecifikáció látható részében, a típusdefiníció után kerültek megadásra. Egy sablon formális paraméterlistájában szereplő átlátszatlan típus ezzel szemben azt jelenti, hogy a *sablonon belül* nincs a típus reprezentációjáról semmilyen információnk. A sablonban csak olyan műveleteket használhatunk a típusértékeken, amik a sablon formális paraméterlistájában a típus mellett megadásra kerültek.

Mint azt a következőkben majd látni fogjuk, ez a párhuzam továbbvihető. Megismerkedünk majd a korlátozott, diszkriminánsos, jelölt, stb. formális paraméter típusokkal, valamint azokkal az esetekkel, amikor a formális paraméter típus specifikációja magában foglalja a típus reprezentációját (valamilyen skalár – diszkrét vagy szám – , tömb, vagy mutató típus) is.

Korlátozott átlátszatlan típusok

Az átlátszatlan típus formális paraméter fenti definícióját kiegészíthetjük a `limited` kulcsszóval.

```
generic
  type T is limited private;
...
```

Az ilyen formális paramétereknek megfeleltethetünk korlátozott (lásd 3.1.2.) aktuális típusokat is. Ezek jellemzője az, hogy nincs rájuk értelmezve az értékadás és az előre definiált egyenlőségvizsgálat. A sablon csak abban az esetben teljesíti a szerződés modell szerinti kikötéseket, ha benne erre a típusra nézve sem értékadás, sem egyenlőségvizsgálat nincs.[8]

Tekintsük például a nemrég bemutatott `Összefésül` eljárássablon egy lehetséges megvalósítását (ami azt feltételezi, hogy a `"<"` művelet egy teljes rendezés az `Elem` típuson):

11.6. Példa

```
procedure Összefésül( X, Y: in File_Type; Z: in File_Type ) is
  SX, SY: Boolean := True;
  DX, DY: Elem;
begin
```

[8] Pontosabban egyenlőségvizsgálatot akkor nem tartalmazhat, ha a sablon formális paraméterlistájában nincs feltüntetve a kérdéses típusra értelmezett `"="` alprogram paraméterként.

```
  Beolvas(SX,DX,X); Beolvas(SY,DY,Y);
  while SX or else SY loop
    if   SX and then (not SY or else DX < DY) then
      Kiír(Z,DX);
      Beolvas(SX,DX,X);
    elsif SY and then (not SX or else DY < DX) then
      Kiír(Z,DY);
      Beolvas(SY,DY,Y);
    else
      Kiír(Z,DX);
      Beolvas(SX,DX,X);
      Beolvas(SY,DY,Y);
    end if;
  end loop;
end Összefésül;
```

Ehhez az eljáráshoz az `Elem` paramétert nyugodtan definiálhatjuk korlátozott átlátszatlan típusnak:

```
generic
  type Elem is limited private;
  with function "<" ( A, B: Elem ) return Boolean;
  with procedure Beolvas ( SX: in out Boolean; DX: out Elem;
                           X: in File_Type );
  with procedure Kiír ( X: in File_Type; E: in Elem );
procedure Összefésül( X, Y: in File_Type; Z: in File_Type );
```

A szerződés modell azt mondja ki, hogy a `limited` kulcsszóval megjelölt formális paraméter típusoknak mind korlátozott, mind korlátozás nélküli aktuális típusok megfeleltethetők, míg a `limited` nélküli formálisaknak csak korlátozás nélküli aktuálisak.

Diszkriminánssal rendelkező típusok

Egy (korlátozott vagy korlátozás nélküli) átlátszatlan típushoz tartozhat – egy vagy több – diszkrimináns (lásd 3.5.1.) is:

```
generic
  type T ( D1: Integer; D2: Boolean ) is limited private;
...
```

Ilyenkor az aktuális paraméternek is olyan típusnak kell lennie, aminek vannak diszkriminánsai, és a diszkriminánsok száma és típusa is meg kell, hogy egyezzen a formáliséival.

A formális típusparaméter diszkriminánsainak nem lehet feltételezett értéke, de az aktuális típus diszkriminánsainak igen (noha nem kötelező, hogy legyen).

```
generic
  type FT ( I: Integer ) is private;
procedure GP ( X: in FT; Y: in out FT );
...
type T ( X: Integer := 42 ) is null record;
procedure P is new GP(T);
```

Egy diszkrimináns nélküli formális paraméter típusnak meg lehet feleltetni egy diszkriminánsos típust, ha az aktuális minden diszkriminánsához van feltételezett érték rendelve. Az előző példában szereplő T típus például megfeleltethető az alábbi formálisnak:

```
generic
  type FT is private;
```

Definit és indefinit típusok

Az Adában[9] megkülönböztetünk definit és indefinit típusokat.[10] A definit típusok olyanok, amelyek objektum létrehozására is alkalmasak, míg az indefinitek olyanok, amelyek nem. Az indefinit típusok közé tartoznak a feltételezett értékkel nem rendelkező diszkriminánsú típusok, a megszorítás nélküli indexhatárú tömbök, de az osztályszintű típusok (lásd 239 o.) is.

Egy formális sablon paraméter típus megadásánál előírhatjuk, hogy a formális típus segítségével lehessen-e objektumokat létrehozni. Amennyiben egy ún. *ismeretlen diszkriminánst* tüntetünk fel a formális paraméter típus specifikációjában, a sablonban a formális paraméter típust nem használhatjuk objektumok létrehozására. Cserében viszont a sablont indefinit aktuális paraméter típussal is példányosíthatjuk:

```
generic
  type IT (<>) is private;
  type DT is private;
procedure G;

procedure G is
  V: DT;  -- megengedett
  -- W: IT;  tilos
begin
  null;
end G;

procedure P is new G(String,Integer);
```

[9] Az Ada95-ben. Az Ada83-ból az ebben a szakaszban ismertetett lehetőségek hiányoztak – komoly rést ütve ezzel a szerződés modell falán.

[10] Az absztrakt jelölt típusok egyik csoportba sem tartoznak.

A `W` változó deklarációja nem helyes, hiszen `IT` aktuális megfelelője a szerződés modell értelmében lehet indefinit típus is (mint például `String`), amivel változót deklarálni nem lehet. Másrészről viszont a `DT` típusnak nem feleltethetünk meg, csak definit típust aktuálisként: a

```
procedure P is new G(String,String);
```

példányosítás nem megengedett.

Az eddig megismert lehetőségek közül tehát az alábbi `S` a legmegengedőbb a példányosítással (a legtöbb típus használható aktuális paraméterként) és legszigorúbb a sablonnal (a legkevesebb szerkezet használható a sablonban):

```
generic
  type S (<>) is limited private;
```

Jelölt típusok

Azt, hogy egy formális sablon paraméternek csak jelölt típust (lásd 12. fej.) szabad aktuálisként megfeleltetni, a **`tagged`** kulcsszóval jelezzük. A **`tagged`** a **`private`** mellett használandó. A szerkezet tartalmazhat **`limited`** kulcsszót, valamint meghatározott és ismeretlen diszkriminánst is.

```
generic
  type T is tagged private;
```

A `tagged`-et tartalmazó formálisnak csak jelölt típust adhatunk meg aktuálisként, `tagged` nélküli formálisnak viszont akár jelölt, akár jelöletlen típus megfeleltethető.

Osztályszintű típusok: A jelölt típusokkal együtt kerültek bevezetésre az osztályszintű típusok. (Egy osztályszintű típus egy jelölt típus és a belőle kiterjesztéssel létrehozott összes típus típusértékeit tartalmazza: ha `T` egy jelölt típus, `T'Class` a hozzá tartozó osztályszintű típus.) Az osztályszintű típusok indefinitek, ezért, ha azt szeretnénk, hogy egy formális paraméternek osztályszintű típust is meg lehessen feleltetni, lássuk el ismeretlen diszkriminánssal. Az osztályszintű típusok jelölt típusok, így a `tagged` kulcsszó is szerepelhet a formális paraméter specifikációjában.

```
generic
  type O (<>) is private;
  type OT (<>) is tagged private;
package G is end G;

type T is tagged null record;
package P is new G(T'Class,T'Class);
```

Absztrakt típusok: Az absztrakt típusok a jelölt típusok egy csoportját alkotják. (Az absztrakt típusok definíciójában szerepel az `abstract` kulcsszó. Ez jelzi azt, hogy a típusnak lehetnek absztrakt műveletei, vagyis olyanok, amelyeknek a specifikációját a programozó megadta, de törzset csak a típus leszármazottaiban akar hozzákapcsolni.) Az absztrakt típusok jellemzője, hogy ugyanúgy, mint az indefinit típusok, nem alkalmasak objektumok létrehozására. Emiatt a sablon szerződésben külön fel kell tüntetni, ha egy formális paraméter típushoz absztrakt típust is kapcsolni engedünk aktuálisként:

```
generic
  type TA is abstract tagged private;
```

A `TA` típusnak absztrakt és nem absztrakt jelölt típusok is megfeleltethetőek, ezzel szemben az olyan formálisaknak, amelyek a specifikációjukban nem tartalmazzák az `abstract` kulcsszót, csak nem absztraktak. Természetesen a sablonon belül `TA` típussal változót nem deklarálhatunk.

Az alábbi formális paraméter típusnak tetszőleges (definit és indefinit, absztrakt és nem absztrakt, diszkriminánsos és diszkrimináns nélküli, korlátozott és korlátozás nélküli, akár osztályszintű) jelölt típus megfeleltethető:

```
generic
  type Jelölt (<>) is abstract tagged limited private;
```

Származtatott típusok

Lehetőség van arra, hogy a formális paraméter típus az aktuális típus reprezentációját is specifikálja. A formális paraméter típus például kijelölheti lehetséges aktuálisnak egy típus összes származtatottját:

```
type T is ...
...
generic
  type TF is new T;
```

A `TF` formálisnak minden olyan típus megfeleltethető aktuálisként, ami a `T` *jelöletlen* típusból – nulla, egy vagy több lépésben – származtatott:

```
type TT is new T;    -- ez is
type TTT is new TT;  -- ez is
```

A származtatott típusok öröklik a `T` típus primitív műveleteit is, így minden lehetséges aktuális paraméter rendelkezik ezekkel a műveletekkel. A szerződés modell ezért megengedi, hogy a sablonon belül a `T` típus összes primitív műveletét használni lehessen a `TF` típusú objektumokra.

Vigyáznunk kell azonban a következőre: ha egy származott típus (például a `TT`) felüldefiniálja a `T` egy primitív műveletét, akkor hiába a `TT` az aktuális

típusparaméter, a sablonon belül az eredeti, T-hez definiált változat fog lefutni a művelet meghívásának hatására.

11.7. Példa

```
package TP is
  type T is null record;
  procedure M( v: in T );
end;
package body TP is
  procedure M( v: in T ) is begin Put_Line("T"); end;
end TP;

use TP;

generic
  type GT is new T;
procedure GP;

procedure GP is
  v: GT;
begin
  M(v);
end GP;

package TTP is
  type TT is new T;
  procedure M( v: in TT );
end;
package body TTP is
  procedure M( v: in TT ) is begin Put_Line("TT"); end;
end TTP;

use TTP;
procedure P is new GP(TT);
```

A P eljárás csak egy T betűt fog kiírni.

A származtatott típus alakú sablon formális paramétert gyakran arra szokták használni, hogy sablonokat *rekord típussal példányosítsunk.*

Kiterjesztett típusok

A jelölt típusok között is van származtatási reláció, amit kiterjesztésnek nevezünk. Ennek megfelelően egy jelölt típusból kiterjesztéssel képzett összes lehetséges típust megengedő formális paraméter típus sémát is tartalmaz a nyelv:

```
type T is tagged ...
...
generic
  type TF is new T with private;
```

Az ilyen fajtájú formális paraméternél a típusműveletek *dinamikusan kapcsolódnak*. Ha az előző szakaszban bemutatott példát tekintjük:

11.8. Példa

```
package TP is
  type T is tagged null record;
  procedure M( v: in T );
end;

package body TP is
  procedure M( v: in T ) is begin Put_Line("T"); end;
  end TP;

use TP;

generic
  type GT is new T with private;
procedure GP;

procedure GP is
  v: GT;
begin
  M(v);
end GP;

package TTP is
  type TT is new T with null record;
  procedure M( v: in TT );
end;
package body TTP is
  procedure M( v: in TT ) is begin Put_Line("TT"); end;
end TTP;

use TTP;
procedure P is new GP(TT);
```

A `P` eljárás két T betűt fog kiírni.

Az ismeretlen diszkriminánst ebben a szerkezetben is használhatjuk annak a jelzésére, hogy az aktuális típus indefinit is lehet. Így lehetőség nyílik arra, hogy egy osztályszintű típussal végezzük el a példányosítást:

```
type T is tagged null record;
generic
  type FT (<>) is new T with private;
procedure G;
...
procedure P is new G(T'Class);
```

Skalár típusok

Formális paraméterként megadható egy skalár típusosztály jelzése is. Ilyen esetekben az aktuális paraméter egy tetszőleges, az adott típusosztályba tartozó típus lehet. A sablonon belül használhatók a típusosztályhoz tartozó előre definiált műveletek, attribútumok. Az alábbiakban példát mutatunk mindegyik típusosztályra.

Diszkrét típus: Ide tartoznak a felsorolási és egész típusok. Példányosítani lehet például a `Boolean` és `Natural` típusokkal. A sablonban használhatók a `'First`, `'Last`, `'Image`, `'Value`, stb. attribútumok, valamint a `"<"`, `"<="`, stb. relációk.

```
generic
  type T is (<>);
```

Előjeles egész típus: Az előjeles egész típusok egyben diszkrét típusok is: megfeleltethetők aktuálisként a formális diszkrét típusoknak. Ha azonban a formálist is előjeles egész típusnak specifikáljuk, a sablonban használhatjuk a diszkrét típusok osztályára jellemző műveletek mellett az egész típusokra vonatkozó aritmetikai (például `"+"`, `"mod"`, stb.) műveleteket is.

Az előjeles egész típusok osztályába tartozik például az előre definiált `Integer`, `Natural` és `Positive` (al)típus.

```
generic
  type T is range <>;
```

Moduló típus: A moduló típusok az egész típusok egy részhalmazát alkotják. Az előjeles egész típusoktól teljesen függetlenül kell őket kezelni. Egyrészt az előjeles egész típusokat nem lehet megfeleltetni a moduló típusoknak, mert a moduló típusokon, ellentétben az előjeles egészekkel, értelmezve vannak logikai műveletek, amiket a sablon törzsében használhatunk is. Másrészről a moduló típusok sem feleltethetők meg az előjeles egészeknek, mert a moduló típusok esetén a túlcsordulás másképp kezelődik, mint előjeles egész típusoknál. Ezért van szükség erre a sablon formális paraméterre:

```
generic
  type T is mod <>;
```

Lebegőpontos típus: Ebbe az osztályba tartozik például a `Float` típus (valamint minden, a `root_float`-ból származó típus).

```
generic
  type T is digits <>;
```

Közönséges fixpontos típus: A programozó-definiálta közönséges fixpontos típusokat soroljuk ide.

```
generic
  type T is delta <>;
```

Decimális fixpontos típus: Ezek a típusok a fixpontos típusok részhalmazát alkotják, a közönséges fixpontos típusok mellett. A két osztály tagjai azonban nem feleltethetők meg egymásnak. Ennek az az oka, hogy a decimális fixpontos típusokra a fordító speciális (például csomagolt tízes számrendszerbeli) reprezentációt választhat.

```
generic
  type T is delta <> digits <>;
```

Tömb típusok

Ha egy sablonon belül ki szeretnénk használni, hogy egy típusparaméter tömb, a formális paramétert így írhatjuk:

```
generic
  type V is array (Index) of Elem;
  type W is array (Index range <>, Index) of Elem;
```

Az `Index` és `Elem` típusok a sablonon kívül definiált típusok vagy a sablon formális paraméterei lehetnek. A sablonon belül a `V` és `W` típusú objektumokra használhatók a tömb típusosztály attribútumai (mint például `'Range`, `'Length`, stb., lásd 3.4.), és természetesen az indexelés is megengedett.

A formális paraméterben az indexek kijelölésénél csak két lehetőség van. Vagy egy (al)típusnév jelzi az indexintervallumot (és a tömb ebben a dimenzióban *megszorított* lesz), vagy a fenti módon – `range <>` – egy *megszorítatlan indexet* kell írni.

Az aktuális paraméter típusnak meg kell egyeznie a formálissal dimenziószámban, a dimenziók indextípusában, megszorítottságában (vagy mindkettő megszorított, vagy egyik sem), megszorított indexpozíció esetén az indexintervallumban, és persze a tömb komponens típusában. Fontos kiemelni, hogy egy megszorított tömb nem felel meg egy megszorítás nélkülinek, és fordítva.

Álljon itt egy jellemző példa formális paraméter tömb típus használatára. Ez egyben bemutatja, hogy a formális paraméter skalár típusokat gyakran mire használják. (Egy másik példa, a `Maximum_Hely` függvénysablon a fejezet elején szerepelt: lásd 230. o.)

11.9. Példa

```
generic
  type Elem is digits <>;
  type Index is (<>);
  type Mátrix is array (Index range <>, Index range <>)
                                                      of Elem;
package Mátrix_Aritmetika is
  function "*" ( A, B: Mátrix ) return Mátrix;
  ...
end Mátrix_Aritmetika;
package body Mátrix_Aritmetika is ...

type Real is digits 5;
type Mátrix is array (Integer range <>, Integer range <>)
                                                      of Real;
package Real_MA is new Mátrix_Aritmetika(Real,Integer,Mátrix);
```

Az alábbi példát érdemes összehasonlítani az említett maximumhely-kereséssel.

11.10. Példa

```
generic
  type Elem is range <>;
  type Index is (<>);
  type Vektor is array (Index) of Elem;
function Sum ( V: Vektor ) return Elem;

function Sum ( V: Vektor ) return Elem is
  Összeg: Elem := 0;
begin
  for I in V'Range loop
    Összeg := Összeg + V(i);
  end loop;
  return Összeg;
end Sum;
```

Két fontos dologban tér el a kettő egymástól. Az egyik a formális paraméter `Vektor` típus definíciója: a `Sum` esetén csak egy megszorított tömb típussal lehet a példányosítást elvégezni. Ez azt eredményezi, hogy az elkészült `Sum` példányt nem lehet meghívni különböző hosszúságú tömbökre.

A másik eltérés az `Elem` típus megadásában van. A maximumhelykeresés esetén egy privát típus volt, az egyetlen szükséges művelettel ("<") kiegészítve. Az összegzés esetén az `Elem` típust előjeles egésznek választottuk. Így a szükséges művelet ("+") és érték (a zérus) sablon-paraméterkénti átadását megspóroltuk. A típusosztály-paraméter használatával picit rövidebb, egyszerűbb lett a program – viszont veszítettünk az általánosságból. Az összegzést nem tudjuk például lebegőpontos típussal példányosítani. Egy sablon programegység megtervezésénél mindig figyeljünk arra, hogy megtaláljuk az arany középutat az általánosság és az egyszerűség (elegancia, könnyű használhatóság) között.

Aliased komponens típus: Ha a formális tömb paraméter komponens típusát az `aliased` kulcsszóval megjelöljük, – akkor, és csak akkor – aktuálisként olyan tömbtípust használhatunk a példányosítás során, aminek a komponens típusa `aliased` típus:

```
generic
  type V is array (Integer range <>) of aliased Integer;
```

Ha a formális tömb komponens típusa `aliased`, az aktuálisnak nem kötelező annak lennie.

Mutató típusok

Egy sablon formális paramétere lehet mutató típus (lásd 3.6.) is. A `P`-nek megfeleltethető aktuális paraméter a `T` típusra mutató típus kell, hogy legyen.

```
generic
  type P is access T;
```

Hasonlóan az objektumra mutató típusokhoz, használhatunk *alprogramra mutató típusokat*[11] is. Az aktuális típusra ismét megköveteljük, hogy mutató típus legyen és a mutatott típus – a paraméterei számában, típusában és módjában – megfeleljen (mód-kompatibilitás, lásd [2]) a formális paraméter definíciójában szereplő alprogramnak.

```
generic
  type P is access procedure ( X: in out Integer );
```

[11] Ez csak az Ada95-re vonatkozik.

A mutató típusok módosító kulcsszavai: Ha a formális mutató paraméter rendelkezik a `constant` módosítóval, akkor és csak akkor az aktuálisnak is rendelkeznie kell vele. Ha a formális definíciójában szerepel az `all` módosító, az aktuálisében is szerepelnie kell. Ez a második eset visszafelé nem teljesül: ha a formális paraméter definíciójában nem szerepel az `all`, az aktuálisében még szerepelhet.

```
generic
  type PC is access constant T;
  type PA is access all T;
```

11.3.3. Paraméter alprogramok

Az eddigi példákban gyakran szerepeltek formális paraméter alprogramok. (Lásd 230. o., 235. o.) Ezek általában a paraméter típusokat egészítik ki, típusműveleteket definiálva hozzájuk.

A formális sablon paraméter alprogramot a `with` kulcsszó vezeti be, majd egy alprogram szignatúra következik. (A `with` jelzi, hogy az utána következő alprogram nem maga a sablon, hanem csak egy formális paramétere.)

```
generic
  type T is private;
  with function Függvény_Paraméter return T;
  with procedure Eljárás_Paraméter ( X, Y: in out T );
procedure Sablon_Eljárás ( A: in out T );
procedure Sablon_Eljárás ( A: in out T ) is
  X: T := Függvény_Paraméter;
begin
  Eljárás_Paraméter(X,A);
end;
```

A példányosításkor olyan alprogramokat feleltethetünk meg a formális paraméternek, amely paraméterei számában, valamint azok típusában és módjában megegyezik a formális paraméter alprogram szignatúrájával.[12] A paraméterek nevének, altípusának és feltételezett értékének, ha van, nem kell megegyeznie az aktuális és formális paraméterben.

```
procedure Eljárás ( I, J: in out Integer ) is
begin
  if I<J then I := J; else J := I; end if;
end Eljárás;
function Függvény return Natural is
begin
```

[12]Egészen pontosan fogalmazva mód-kompatibilitás szükséges, mint az alprogram-átnevezések esetén.

```
  return 0;
end Függvény;
procedure Példány is
             new Sablon_Eljárás(Positive,Függvény,Eljárás);
```

Az alábbi példa egy gyakori hibát szemléltet:

```
with Text_IO, Ada.Integer_Text_IO;
use  Text_IO, Ada.Integer_Text_IO;
procedure Összead is
  generic
    type T is range <>;
    with procedure Kiír ( E: in T );
  procedure Összeadást_Kiír ( A, B: in T );
  procedure Összeadást_Kiír ( A, B: in T ) is
  begin
    Kiír(A); Put(" + "); Kiír(B); Put(" = "); Kiír(A+B);
  end Összeadást_Kiír;
  procedure Integer_O_K is new Összeadást_Kiír(Integer,Put);
...
```

A példányosítás hibás: igaz, hogy az `Ada.Integer_Text_IO.Put` eljárást meg lehet hívni egy `Integer` paraméterrel, de a szignatúrájában van még két paraméter (feltételezett értékkel ellátva). Így a formális paraméter alprogramnak, aminek csak egy paramétere van, nem feleltethető meg. A probléma megoldása:

```
procedure Integer_Kiír ( I: in Integer ) is
begin
  Put(I);
end Integer_Kiír;
procedure Integer_O_K is
                 new Összeadást_Kiír(Integer,Integer_Kiír);
```

Formális paraméter alprogram lehet *operátor* is. Erre a maximumhely-keresést hozhatjuk fel példának, ahol a `"<"` műveletet adtuk át paraméterként.

Ez pedig egy példa arra, hogy az *aktuális paraméter* egy *attribútum* is lehet:

```
generic
  type T (<>) is limited private;
  with function toString ( E: T ) return String;
procedure Kiír ( E: in T );
...
procedure B_Kiír is new Kiír(Boolean,Boolean'Image);
```

Formális paraméter alprogramok feltételezett értéke

Két lehetőség van arra, hogy formális alprogramparamétert feltételezett értékkel lássunk el. Az első lehetőséget az előző példán mutatjuk meg. Az alprogram szignatúrája után az `is` kulcsszót és egy nevet kell írni:

```
generic
  type T (<>) is limited private;
  with function toString ( E: T ) return String is T'Image;
procedure Kiír ( E: in T );
...
procedure B_Kiír is new Kiír(Boolean);
```

Amennyiben a példányosításkor a feltételezett értékkel ellátott paraméter alprogramnak nem feleltetünk meg aktuálist, de van a feltételezett értékkel, ami egy név, és a formális paraméternek megfelelő szignatúrával rendelkező alprogram, akkor az az alprogram lesz az aktuális paraméter. Fontos szabály, hogy a sablon deklarációjának helyén már ismertnek kell lennie annak az alprogramnak, amit a feltételezett értéknek írunk. Ez a sablon deklaráció hibás:

```
function Image ( B: Boolean ) return String renames
                                              Boolean'Image;
generic
  type T (<>) is limited private;
  with function toString ( E: T ) return String is Image;
procedure Kiír ( E: in T );
```

Ez viszont helyes lesz:

```
procedure Default_Akció ( X, Y: in Natural );
generic
  with procedure Akció ( X, Y: in Natural ) is Default_Akció;
procedure Mindre_Végrehajt;
...
procedure Saját_Mindre_Végrehajt is new Mindre_Végrehajt;
```

A másik, gyakrabban használható lehetőség egy sablon formális paraméter alprogram feltételezett értékének specifikálására az, ha a formális paraméter alprogram szignatúrája után az `is <>` szintagmát írjuk. A maximumhely-keresést hozva fel példaként:

```
generic
  type Elem is private;
  type Index is (<>);
  type Vektor is array (Index range <>) of Elem;
  with function "<" ( A, B: Elem ) return Elem is <>;
function Maximum_Hely ( V: Vektor ) return Elem;
```

A példányosításnál nem szükséges aktuálist megfeleltetni a "<" függvénynek, ha *a példányosítás helyén* ezzel a névvel (és persze a formális paraméter alprogram specifikációjának megfelelő) alprogram látható:

```
...
function Maxhely is new Maximum_Hely(Integer,Positive,Vektor);
```

Természetesen több formális paraméternek is lehet feltételezett értéke, ilyenkor a példányosításnál sokat segíthet a névvel jelölt paraméter-megfeleltetés lehetősége.

11.3.4. Csomagparaméterek

A negyedik sablon paraméter forma a formális paraméter csomag. Használata sokszor kényelmesebbé teszi a sablonokkal való programozást. Gyakori, hogy egy sablon egy meglévő másik sablon szolgáltatásait egészíti ki, vagy egyszerűen csak a másik sablon által definiált absztrakcióra épít. Nézzünk erre egy példát. A 245. oldalon megmutattuk, hogyan nézhet ki egy alapvető mátrixműveleteket (például összeadást, szorzást, skalárral való szorzást) definiáló sablon csomag. Tegyük fel, hogy most egy olyan sablont szeretnénk írni, ami ezekre az egyszerű aritmetikai operátorokra építve bonyolultabb műveleteket (például mátrix invertálást, háromszögekre bontást, stb.) valósít meg. Így deklarálhatnánk:

```
generic
  type Elem is digits <>;
  type Index is (<>);
  type Mátrix is array (Index range <>, Index range <>)
                                                of Elem;
  with function "*" ( A, B: Mátrix ) return Mátrix is <>;
  ...
package Összetett_Mátrix_Mûveletek is
  function Inverz ( M: Mátrix ) return Mátrix;
...
```

Látható, hogy a `Mátrix_Aritmetika` sablon összes paraméterét, sőt, az összes benne definiált műveletet fel kell sorolnunk új sablonunk paraméterlistájában. A példányosítás így történhet:

```
...
package Real_MA is new Mátrix_Aritmetika(Real,Integer,Mátrix);
use Real_MA;
package Real_ÖMM is new
                Összetett_Mátrix_Mûveletek(Real,Integer,Mátrix);
```

Itt valamennyire segített rajtunk az, hogy a paraméter alprogramok feltételezett értékét használhattuk, de még így is sokkal bonyolultabb kód született, mintha a következő, elegáns megoldást használtuk volna:

```
generic
  with package Aritmetika is new Mátrix_Aritmetika (<>);
package Összetett_Mátrix_Mûveletek is
  function Inverz ( M: Aritmetika.Mátrix ) return
                                        Aritmetika.Mátrix;
...
```

És a példányosítás:

```
package Real_MA is new Mátrix_Aritmetika(Real,Integer,Mátrix);
package Real_ÖMM is new Összetett_Mátrix_Mûveletek(Real_MA);
```

Ebben a második példában egy csomagot adtunk át paraméterként. A formális paraméter csomag specifikációja tartalmazza egy *sablon csomag* nevét: az aktuális paraméter csomag ennek a sablon csomagnak kell, hogy *egy példánya* legyen.

A formális paraméter csomag (itt `Aritmetika`) neve egy csomagot (nem egy sablon csomagot) jelent az `Összetett_Mátrix_Mûveletek` sablonban. Komponenseire is hivatkozhatunk, például a `Aritmetika."+"` műveletre, sőt, akár a formális paraméter csomag formális paramétereire is, például az `Aritmetika.Mátrix` típusra. A minősítés feloldása is lehetséges, az alábbi módon:

```
generic
  with package Aritmetika is new Mátrix_Aritmetika (<>);
package Összetett_Mátrix_Mûveletek is
  use Aritmetika;
  function Inverz ( M: Mátrix ) return Mátrix;
...
```

A `(<>)` jelzés a formális paraméter csomag specifikációjában azt jelenti, hogy a formális paraméternek bármilyen olyan csomagot megfeleltethetünk aktuálisként, ami az előtte álló sablon (jelen esetben a `Mátrix_Aritmetika`) egy példánya. Arra is lehetőség van, hogy ennél szigorúbb követelményt szabjunk a példányosításra. Tegyük fel – a példánknál maradva, – hogy szükségünk van az `Összetett_Mátrix_Mûveletek` sablon megvalósításához még egy sablon csomagra, amiben a `Float` típushoz írott műveleteink találhatók:

```
generic
  type T is digits <>;
package Float_Mûveletek is
  -- T-n értelmezett eljárások, függvények
end;
```

Az Összetett_Mátrix_Műveletek formális paraméterlistájában, ez az alábbi módon adható meg:

```
generic
  with package Aritmetika is new Mátrix_Aritmetika (<>);
  with package Elem_Mûveletek is new Float_Mûveletek (<>);
...
```

Igen ám, de mi van akkor, ha olyan aktuálist feleltetünk meg az Aritmetika és Elem_Mûveletek paramétereknek, amelyek különböző lebegőpontos típusokkal lettek példányosítva a megfelelő sablon csomagból? Ezt kizárandó, egy formális paraméter csomag specifikációjában megadhatjuk, hogy milyen paraméterrel példányosított csomag lehet az aktuális:

```
generic
  with package Aritmetika is new Mátrix_Aritmetika (<>);
  with package Elem_Mûveletek is
                new Float_Mûveletek (Aritmetika.Elem);
package Összetett_Mátrix_Mûveletek is
  use Aritmetika, Elem_Mûveletek;
  function Inverz ( M: Aritmetika.Mátrix ) return
                                   Aritmetika.Mátrix;
...
```

Így már biztosított, hogy az Elem_Mûveletek aktuálisát ugyanazzal a típussal szabad csak példányosítani, mint amivel az Aritmetika aktuálisát:

```
package Real_MA is new Mátrix_Aritmetika(Real,Integer,Mátrix);
package FM is new Float_Mûveletek(Real);
package Real_ÖMM is
            new Összetett_Mátrix_Mûveletek(Real_MA,FM);
```

Azon paraméter csomagok formális paramétereit, amelyeknek specifikációjában nem (<>) szerepel, nem használhatjuk a sablonon belül: helyettük a megjelölt aktuálisakra lehet hivatkozni: példánkban az Elem_Mûveletek T típusa helyett az Összetett_Mátrix_Mûveletek sablonon belül az aktuális Aritmetika.Elem típus használandó.

Sablon formális paraméterlistájába is írhatunk use záradékot. Ez különösen akkor hasznos, ha egy sablon alprogram szignatúrájának specifikálásához olyan nevet szeretnénk használni, ami egy sablon paraméter csomagban van definiálva:

```
generic
  with package Aritmetika is new Mátrix_Aritmetika (<>);
  use Aritmetika;
function Inverz ( M: Mátrix ) return Mátrix;
...
```

Az Inverz szignatúrájában a Mátrix az Aritmetika.Mátrix típust jelenti.

11.4. Példák sablonok használatára

Az előző szakaszokból megismerhettük a sablonok használatára vonatkozó legfontosabb szabályokat, a sablonok főbb alkalmazási területeit. Jó néhány érdekes dolog azonban nem került még szóba. Ezek közül fogunk most néhányat – példaprogramokon keresztül – bemutatni.

11.4.1. Sablonok az előre definiált könyvtárakban

Az Ada előre definiált könyvtárai sok sablon programegységet tartalmaznak. A legtöbb ilyen programegység a be- és kimenet (`Ada.Direct_IO`, `Ada.Sequential_IO`, stb., lásd A.5. Függelék) és a matematikai könyvtárak (például az `Ada.Numerics.Generic_Complex_Types`, stb., ld. G. Függelék) témaköréhez kapcsolódik, de vannak előre definiált sablonok például a memóriagazdálkodással (`Ada.Unchecked_Deallocation`) és ellenőrizetlen típuskonverziókkal (`Ada.Unchecked_Conversion`) kapcsolatban is.

Az egyik, talán leggyakrabban használt sablon az `Ada.Text_IO` csomagban definiált `Integer_IO` sablon csomag. Segítségével valamilyen (paraméterként megadható) előjeles egész típus értékeit tudjuk szövegfájlokból (vagy billentyűzetről) beolvasni és szövegfájlokba (vagy a képernyőre) kiírni.

```
with Text_IO;
procedure Sokáig_Futó_Program is
  package Pos_IO is new Text_IO.Integer_IO(Positive);
begin
  Pos_IO.Put(42);
  Text_IO.New_Line;
end Sokáig_Futó_Program;
```

Ezt a sablont `Integer` típusú értékekre olyan gyakran szokták példányosítani, hogy az Ada95 nyelv tervezői jónak látták a megfelelő nem sablon változatot is hozzávenni a előre definiált könyvtárakhoz:

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
...
```

11.4.2. Sablonban, illetve sablonként megvalósított típusok

Csomagokat használhatunk arra, hogy objektumokat vagy típusokat valósítsunk meg bennük. Erre mutatott példát 7.7.2. és 7.7.3. Mi történik akkor, ha ugyanezeket a technikákat sablon csomagokkal alkalmazzuk?

Ha egy sablon csomagban egy típust valósítunk meg, a sablon példányosításával újabb és újabb példányokat hozhatunk létre az adott típusból.

Gondoljunk vissza a `Vermek` sablon példára. (Lásd 228. o.) Ezen sablon csomag segítségével különböző `Verem` típus példányokat hozhatunk létre: a példányosításhoz meg kell adni a létrehozandó `Verem` típus példány komponens típusát és maximális méretét. Minden `Verem` típus példány segítségével tetszőleges számú verem objektumot definiálhatunk:

```
with Vermek;
procedure A is
  package Int_42_Vermek is new Vermek(Integer,42);
  package Bool_24_Vermek is new Vermek(Boolean,24);
  use Int_42_Vermek, Bool_24_Vermek;
  V, W: Int_42_Vermek;
...
```

Sokszor nincs arra szükségünk, hogy „akárhány" verem objektumot definiálhassunk. Ilyenkor a sablon csomagban elég egy objektumot megvalósítani:

11.11. Példa

```
generic
  type Elem is private;
  Max_Méret: in Positive;
package Verem is
  procedure Push ( E: in Elem );
  procedure Pop ( E: out Elem );
  ...
end Verem;
package body Verem is
  Tartalom: array (1..Max_Méret) of Elem;
  Verem_Tetô: Natural := 0;
  procedure Push ( E: in Elem ) is
  begin
    -- feltesszük, hogy nincs tele a verem
    Verem_Tetô := Verem_Tetô + 1;
    Tartalom(Verem_Tetô) := E;
  end Push;
  ...
end Verem;
```

A sablon példányosításával csomagokat gyárthatunk, és mindegyik csomag egy objektumnak feleltethető meg. Ezek az objektumok mind vermek ugyan, de különböző komponens típussal és maximális mérettel rendelkezhetnek.

```
with Verem;
procedure B is
  package V is new Verem(Integer,42);
```

```
  package W is new Verem(Boolean,24);
begin
  V.Push(1088);
  V.Push(6-8);
  ...
end;
```

Mivel a `Verem` sablonból létrehozott példányok objektumoknak feleltethetők meg, a sablon maga „objektum-gyárnak”, azaz típusnak tekinthető.

11.4.3. Üres sablon csomag, mint absztrakció

A sablon formális paraméter csomagok felhasználásának egy módjára mutatunk most példát. Ami az igazán érdekes benne az az, hogy mire használható egy teljesen üres sablon csomag.

11.12. Példa

```
package Fa_Bejárás is

  generic
    type Fa (<>) is limited private;
    with function Bal  ( F: Fa ) return Fa;
    with function Jobb ( F: Fa ) return Fa;
    with function Üres ( F: Fa ) return Boolean;
  package Fa_Absztrakció is end;

  type Sorrend is ( Preorder, Inorder, Postorder );

  generic
    with package Fa_Példány is new Fa_Absztrakció(<>);
    with procedure Mývelet_A_Gyökéren ( F: in Fa_Példány.Fa );
    use Fa_Példány;
  procedure Bejár ( F: in Fa; S: in Sorrend := Inorder );

end Fa_Bejárás;
```

A `Fa_Bejárás` csomag definiálja a `Bejár` műveletet. Ez a `Bejár` valamilyen fa típusú objektumot vár paraméterül. Ezt a szintén paraméterként megadott `S` bejárási stratégia szerint bejárja, minden csomóponton elvégezve a `Mývelet_A_Gyökéren` sablon paraméter eljárást.

A `Fa_Absztrakció` sablon csomag definiálja azt, hogy mit is jelent a fa fogalma: olyan rekurzív típus, ami, ha nem üres, gyökérelemet, baloldali és jobboldali részfát tartalmaz. A `Bejár` eljárás szempontjából a fa implementációja teljesen lényegtelen. Ha egy adatszerkezet teljesíteni tudja a

Fa_Absztrakció-ban definiált követelményeket, akkor azon az adatszerkezeten értelmezhető (és, mint a következőkben látni fogjuk, két példányosítással létre is hozható) a Bejár is.

11.13. Példa

```
package body Fa_Bejárás is
  procedure Bejár ( F: in Fa; S: in Sorrend := Inorder ) is
  begin
    if S=Preorder then Mývelet_A_Gyökéren(F); end if;
    if not Üres(Bal(F)) then Bejár(Bal(F),S); end if;
    if S=Inorder then Mývelet_A_Gyökéren(F); end if;
    if not Üres(Jobb(F)) then Bejár(Jobb(F),S); end if;
    if S=Postorder then Mývelet_A_Gyökéren(F); end if;
  end Bejár;
end Fa_Bejárás;
```

Ez a sablon csomag törzsének egy lehetséges megvalósítása.

Most pedig készítünk egy olyan adatszerkezetet, ami implementálja a Fa_Absztrakció-t. A fa sorfolytonos ábrázolásban álljon az 1..Utolsó számokból:

11.14. Példa

```
Utolsó: Positive := 20;
function Bal_Gyerek ( I: Positive ) return Positive is
  begin return 2*I; end;
function Jobb_Gyerek ( I: Positive ) return Positive is
  begin return 2*I+1; end;
function Kilóg ( I: Positive ) return Boolean is
  begin return I>Utolsó; end;
```

Választhattunk volna egészen más reprezentációt, például valamilyen rekordokat tartalmazó láncolt adatszerkezetet, és készíthettünk volna arra megfelelő műveleteket: a lényeg az, hogy az alábbi példányosítással igazoljuk, hogy amit írtunk, az tényleg egy fa.

```
package Index_Fa is new Fa_Bejárás.Fa_Absztrakció (
          Positive, Bal_Gyerek, Jobb_Gyerek, Kilóg );
```

Ezek után nincs más hátra, mint megadni, hogy milyen műveletet kell a fa csomópontjain elvégezni, és ennek megfelelően példányosítani a Bejár eljárást. A példánkban egyszerűen kiírjuk a csomópontokat a képernyőre:

```
procedure Gyökeret_Kiír ( I: Positive ) is
  begin Text_IO.Put(Positive'Image(I)); end;
procedure Fát_Kiír is
            new Fa_Bejárás.Bejár(Index_Fa,Gyökeret_Kiír);
```

A kapott eljárást például így hívhatjuk meg:

```
Fát_Kiír(1,Preorder);  Text_IO.New_Line;
Fát_Kiír(4);           Text_IO.New_Line;
Fát_Kiír(2,Postorder); Text_IO.New_Line;
```

11.4.4. Kivételek sablon formális paraméterként

Az `Ada.Exceptions` csomagban definiált típusok és műveletek a kivételeket „hagyományos” objektumokra képzik le. Segítségükkel lehetőség nyílik arra, hogy egy sablonnak kivétel(eke)t adjunk meg formális paraméterkét. (Lásd 138. o.)[13] A `Max` függvény alábbi megvalósítása álljon itt példaként:

11.15. Példa

```
with Ada.Exceptions;
use Ada.Exceptions;
generic
  type T is private;
  type Index is (<>);
  type Vektor is array (Index range <>) of T;
  Üres_Vektor: Exception_Id := Constraint_Error'Identity;
  with function "<" ( A, B: T ) return Boolean is <>;
function Max ( V: Vektor ) return T;

function Max ( V: Vektor ) return T is
begin
  if V'Length = 0 then Raise_Exception(Üres_Vektor); end if;
  declare
    MH: Index := V'First;
  begin
    for I in V'Range loop
      if V(MH) < V(I) then MH := I; end if;
    end loop;
    return V(MH);
  end;
end Max;
```

11.5. Feladatok

11.1. Valósítsa meg a `Verem`, illetve `Sor` típusokat sablon csomagként, illetve sablon csomagban, majd írjon olyan eljárást, amely egy sort megfordít egy verem segítségével!

[13] Ez az Ada83-ban nem volt megvalósítható.

11.2. Írjon sablon bináris keresést! A feladat megkeresni egy egész indextípusú rendezett tömbben egy értéket. Hogyan módosul a program, ha nem egy tömbben, hanem egy paraméterként megadható, egészeken értelmezett monoton függvény értékei között kell keresni? És ha az indextípus – illetve az értelmezési tartomány – tetszőleges diszkrét típus lehet? Ha a keresett elem nem létezik, a keresés váltson ki egy paraméterként megadható kivételt!

11.3. Készítsen – egymásra épülő – félcsoport, csoport és gyűrű absztrakciót!

11.4. Valósítson meg egy – 3 dimenziós – térbeli pontokat leíró típust! Műveletei: eltolás, valamint valamelyik koordináta-tengelyre vonatkozó, illetve középpontos tükrözés és forgatás. A pontok koordinátái és a forgatási szögek lebegőpontos típusúak legyenek; ezeket a típusokat sablon paraméterként kell megadni.

Ezután – a pont típusra, és az azt tartalmazó sablon csomagra építve – készítse el az egyenesek leírására szolgáló típust. Pontok tengely körüli forgatása és tengelyes tükrözése mellett írjon hozzá olyan műveleteket is, amelyek meghatározzák egy egyenes és egy pont, illetve két egyenes egymáshoz viszonyított helyzetét, távolságát.

12. Fejezet

Objektum-orientált programozás

A következő fejezetben áttekintjük az objektum-orientált programozás alapfogalmait, és azt, hogyan támogatja az elveket az Ada95 programozási nyelv.

12.1. Az objektum-orientált tervezés

A jó minőségű szoftver alapvető kritériumai a *karbantarthatóság* és az *újrafelhasználhatóság*. (ld. Bevezetés)

Megfigyelhetjük, hogy a professzionális programok, programrendszerek működése, továbbfejlődése során, az elvégzendő tevékenységek meglehetősen gyakran változnak. Ezért, ha a rendszer felépítését a funkciói alapján határozzuk meg, akkor egy-egy ilyen változtatás akár teljesen fel is boríthatja a rendszer szerkezetét.

Ugyanakkor azonban úgy tűnik, hogy az adatokat alapvetően a változások nem érintik, ezek csak a rendszer összetevőire vannak hatással – egy hallgatói nyilvántartási rendszer például változatlanul a diákok adataiból indul ki, még ha az ösztöndíjak, vagy tandíjak kiszámítási szabályai meg is változnak. Ez a legfőbb érv amellett, hogy a rendszerek struktúráját az általuk kezelt adatokra, illetve valójában a szereplő adattípusokra alapozzuk.

Egy szoftver rendszerben természetesen mind az adatok, mind a tevékenységek helyet kell kapjanak, a kérdés csak az, hogy melyiket tekintjük elsődlegesnek a rendszer felépítése szempontjából.

Az *objektum-orientált tervezés* – ahogy az elnevezés is mutatja – olyan szoftver-konstrukciós módszer, amely a programrendszert az általa használt adatok, a valós világ modellezésével kapott objektumok alapján építi fel. Mottóként szokták ([17]) megfogalmazni: „Ne azt kérdezzük először, hogy *mit* csinál a rendszer, hanem azt, hogy *mivel* csinál valamit”.

Ha objektum-orientáltan tervezünk, akkor először a rendszer objektumait vizsgáljuk, megpróbáljuk általánosan leírni ezeket, s csak ezután adjuk meg a programrendszer legfelső funkciójának pontos leírását. Ez a „bottom-up" vagyis *alulról felfelé történő tervezés* lényege. Ez azt jelenti, hogy az elérhető komponensek szintjéről indulunk, és azokból építkezünk. Vagyis: ahelyett, hogy arra törekednénk, hogy az aktuális problémára a lehető legoptimálisabb megoldást adjuk, megkísérlünk egy jó megoldást adni, azzal minimalizálva az erőfeszítést, hogy az előző eredményekből építkezünk, és arra törekszünk, hogy az általánosság lehető legmagasabb szintjét érjük el, megkönnyítve ezzel a további fejlesztéseket.

Mit tekintünk *objektum*nak? Az objektum-orientált szemléletben az objektumok a modellezendő valós világ egy-egy önálló egységét jelölik. Az objektumokat meghatározza *belső állapot*uk és a *nekik küldhető üzenetek*re való reagálásuk. Ez a „reagálás" jelentheti azt, hogy megváltozik az objektum belső állapota, és(vagy) az objektum elvégez valamilyen „műveletet", kijelez a külső világ számára valamilyen értéket, üzenetet küld más objektumoknak, létrehoz, töröl, másol, illetve mozgat más objektumokat stb.

A valós világ leírásakor általában az objektumokat bizonyos tulajdonságaik azonossága alapján *osztály*okba soroljuk. Például megkülönböztetjük az élőlényeket az élettelenektől, s az élőlényeket különböző alosztályokba csoportosíthatjuk – tekintsük például erre az állatrendszertani leírást: amikor egy ismeretlen állatot szeretnénk besorolni, eldöntjük, hogy a puhatestűek, az ízeltlábúak, a gerincesek stb. törzsébe tartozik-e, majd, ha például úgy találjuk, hogy ízeltlábú, akkor megvizsgáljuk, hogy a rákok, rovarok vagy pókszabásúak osztályába esik-e, s ha rovar, akkor szárnyas vagy szárnyatlan. A szárnyas rovaroknak is sok alosztálya van: a pillangók, legyek, bogarak stb.

Minden kategóriának számos alkategóriája lehet, s ezek alatt az alkategóriák alatt is további alkategóriák vannak. Minden szint speciálisabb, mint az őt megelőző, ugyanakkor az is igaz, hogy ha egy tulajdonságot (jellemzőt) definiáltunk, akkor minden alosztály ez alatt a szint alatt tartalmazza ezt a jellemzőt. Vagyis, ha például egyszer már azonosítottunk egy élőlényt, mint a lepkék rendjébe tartozót, azután már nem szükséges külön kiemelnünk, hogy ízeltlábú – ez következik abból, hogy a lepkék az ízeltlábúak közé tartoznak.

A fentieknek megfelelően más esetekben is általánosan leírhatjuk, hogy mi jellemzi az egyes osztályokat, milyen alosztályaik vannak stb..

Az alosztályok létrehozásánál mindig támaszkodunk annak az osztálynak a tulajdonságaira, amelyiknek az új alosztály része. Ezt úgy is mondhatjuk, hogy az *alosztály örökli az ősosztály tulajdonságait.* Természetesen az ősosztály is lehet, hogy egyúttal egy másik osztály alosztálya, vagyis tulajdonságainak egy részét szintén örökölte. Ezáltal az alosztályok leírása leegyszerűsödik, hiszen az öröklött jellemzőket nem kell mindig leírni, egyszerűen újra felhasználjuk a már adott osztályokat.

Ekkor az új alosztályt a kiindulásul vett osztály *örökös*ének, illetve a kiindulásul vett osztályt az új osztály *szülő*jének nevezzük. Az öröklést tehát úgy tekinthetjük, mint osztályok között egy alosztályképzési műveletet.

Így, ha például a `Kör` osztály az `Alakzat` osztály örököseként jött létre, akkor a `Kör` objektumaira igaz, hogy azok egyúttal `Alakzat` típusúak is. Ez azt jelenti, hogy egy `Kör` típusúként deklarált `b` objektumot tekinthetünk `Alakzat` típusúként is. Fordítva természetesen nem igaz az állítás, egy `Alakzat`-beli objektum nem feltétlenül esik egyúttal a `Kör` által leírt altípusba is. Ezt a lehetőséget, nevezetesen, hogy egy változó több osztály objektumaként is tekinthető, nevezzük a változók *polimorfizmus*ának (többalakúságnak).

A programrendszer a valós világot modellezi. Az osztályok leírásához a legmegfelelőbb eszközt az *absztrakt adattípus*ok jelentik: jellemezni kell az objektumok *értékhalmazát* és rögzíteni az *elérhető műveletek halmazát*, definiálva ezek absztrakt tulajdonságait. Így egy típusspecifikációt adunk meg, amely még nem törődik a reprezentációval, csak azt határozza meg, hogy milyen szolgáltatásokat nyújt a külvilág felé. A reprezentációt, és a műveletek megfelelő implementációját elrejtjük, ezt *egységbezárás*nak is nevezik.

Az előbb vázolt alosztályképzési mechanizmus voltaképpen olyan eszközt igényel, amelynek segítségével moduljaink egyszerre zártak és nyíltak. Zártak, hiszen önmagukban leírnak egy osztályt – mondjuk az ízeltlábúak, vagy a grafikus alakzatok osztályát – , a megfelelően definiált műveletekkel, de ugyanakkor nyíltak, s ezzel lehetővé teszik, hogy segítségükkel az eredeti osztály alosztályait, specializációit hozzuk létre, – például a rovarok, a sokszögek, vagy a körök osztályait – úgy, hogy a leírásnál nem kell mindent előlről kezdeni, felhasználhatjuk az eddigi osztályokat.

A zárt modulok által nyújtott, az interfészben leírt szolgáltatásokat más modulok egyszerűen felhasználhatják, s ha lehetőségünk van rá, hogy ugyanezen modulokat nyíltként használjuk, ez eszközt ad kezünkbe az öröklődés megvalósítására.

Az altípusképzés során megengedjük, hogy az új osztályokban az egyes műveletek implementációja különbözzön az ősosztálybeli implementációtól – például az `Alakzat`-ok osztályának lehet egy `Kirajzol` művelete, amely egy általános grafikus alakzat képernyőre való kirajzolását eredményezi, jogos elvárásunk, hogy ennek az osztálynak pl. a köröket, vagy a sokszögeket leíró örököseinél a `Kirajzol` művelet egy kört, illetve sokszöget rajzoljon ki. Dinamikus összekapcsolás kell lehetőséget adjon arra, hogy a változó aktuális típusa – futás közben – határozza meg a művelet végrehajtandó implementációját.

Lehetőséget kell biztosítani arra is, hogy az egyes alosztályokban bővítsük a lehetséges műveletek halmazát.

Egy programozási nyelvet tehát akkor nevezhetünk objektum-orientált tervezést támogató nyelvnek, ha lehetővé teszi a következőket:

Absztrakt adattípusok kezelése, ezen belül:

- **Objektumok.** Entitások, melyeknek belső állapotuk és azt megvalósító struktúrájuk van.
- **Műveletek.** Műveletek az objektummal, melyek segítségével el lehet érni vagy módosítani lehet annak állapotát.
- **Egységbezárás.** A felhasználó csak az absztrakt interfészt látja, a megvalósítás részleteit el lehet rejteni.

Öröklődés. Egy újabb absztrakt adattípus megalkotásánál felhasználjuk a korábban már megírt adattípusaink tulajdonságait, anélkül, hogy megsértenénk annak a megvalósítását. Van lehetőség arra, hogy az öröklés során bizonyos műveleteket megváltoztassunk az adott specializációnak megfelelően, illetve bővíthetjük a művelethalmazt.

A továbbiakban áttekintjük, hogyan valósították meg az Ada95-ben az előző követelményeket.

Az Ada95 már meglévő programozási nyelv kiterjesztése, így az egyik fő szempont az Ada83-mal való kompatibilitás volt, ezért továbbra is lehetőség van hagyományos összetett adattípusok (rekordok, tömbök) deklarálására is az osztályok deklarációja mellett. Ennek következtében az Ada95 programozási struktúrája – hasonlóan a C++-hoz – nem szigorúan objektum-orientált, ellentétben a Java-val vagy az Eiffellel, ahol összetett típust csak az osztályokkal lehet megvalósítani. Az osztályok (absztrakt adattípusok) leírására – a C++-szal ellentétben – nem egy teljesen új fogalmat, illetve struktúrát vezettek be, hanem a már meglévő rekordfogalmat bővítették ki. Ezek a speciális, ún. *jelölt* rekordok csak az adott osztályok attribútumait tartalmazzák, az osztályok műveleteit külön kell megadni.[1] A láthatóság szabályozása a csomagok segítségével lehetséges.

12.2. Öröklődés

Az öröklődés az Ada83-ból ismert típusszármaztatás fogalmára épül. A felhasználó által definiált típusból származtatott típus, ahogy a 3.1.2. alfejezetben láttuk, ha egy csomag specifikációs részében definiáljuk, akkor a predefinit típusokból való származtatáshoz hasonlóan örökli a bázistípus műveleteit, és definiálhatunk hozzá újabb műveleteket.

Az Ada95-ben a származtatott típust új komponensekkel is kibővíthetjük, a *jelölt* (`tagged`) típusok segítségével. Az elgondolt implementációs modell az, hogy minden objektumnak lesz egy *címke* (tag) komponense,

[1] Voltaképpen az Ada95-ben éppen ezért nem is beszélhetünk a hagyományos értelemben absztrakt adattípusokról.

ami egy mutató a statikusan allokált és szerkesztési időben inicializált típus-leíróra. A típus-leíró tartalmazza a típus ún. *primitív művelet*ei kódjának a címét (esetleg egyéb információkat is, amelyek például megkönnyítik a tartalmazási ellenőrzéseket).

Egy rekord típus (vagy egy átlátszatlan (`private`) típus), amelyiknek a definíciójában a `tagged` kulcsszó szerepel, egy *jelölt* típus lesz, s ha ebből származtatunk, akkor a típus kiterjeszthető, hozzá további komponensek definiálhatóak. A típushoz ún. *primitív művelet*eket is definiálhatunk - ezek öröklődnek, illetve ezek definiálhatóak úgy felül a későbbi leszármazottaknál, hogy futási idejű kiválasztásuk lehetséges, ha a típusdefiníció egy csomag specifikációs részében szerepel.

A *származtatott típus*t az *ős típus kiterjesztésének*, vagy egyszerűen *típus-kiterjesztés*nek nevezzük. Minden típus-kiterjesztés maga is egy jelölt típus. Egy rekord kiterjesztés definiálható egy (esetleg üres) rekord kiterjesztési résszel.[2]

Egy jelölt típus *primitív műveletei* jelentik a típus első „fagyasztási pontját" – a típus-leíró ennek alapján készül el. Ha egy jelölt típust nem egy csomag specifikációban deklaráltunk, akkor lehetetlen új primitív műveleteket adni ehhez a típushoz, bár örökölhet primitív műveleteket, és ezek felüldefiniálhatóak.

A következő példában a típuskiterjesztés legegyszerűbb változatát vizsgáljuk meg, tekintsünk egy részletet a geometriai objektumok hierarchiájából!

12.1. Példa

```
with ...; use ...;
      -- grafikus lehetőségeket tartalmazó csomag
package Geometriai_Alakzatok is
 type Alakzat is tagged record
   Szín: Képernyô_Szín := Fehér;
  end record;
-- Az Alakzat típus primitív müveletei:
  procedure Kirajzol(O: in Alakzat); -- nem csinál semmit,
     -- üres a törzse
  function Mi_a_Színe(O: in Alakzat) return Képernyô_Szín;

 type Pont is new Alakzat with record
    X_Kord, Y_Kord: Float;
  end record;
-- A Pont típus primitív müveletei az öröklöttek,
-- illetve az átdefiniáltak,
-- valamint az újonnan bevezetettek:
```

[2] Egy átlátszatlan kiterjesztés, ami a rekord kiterjesztés egy parciális nézete, deklarálható egy csomag látható részében, vagy egy sablon formális részben.

```
   procedure Kirajzol(P: in Pont);
      -- kirajzolja a pontot
   procedure Elmozdít(P : in Pont; X1,Y1:Float);
      --áthelyezi a pontot

  type Kör is new Pont with record
     R: Float := 0.0;
   end record;
 -- A Kör típusra átdefiniáljuk a Kirajzol primitív műveletét:
   procedure Kirajzol(C: in Kör);     -- kirajzolja a kört

 end Geometriai_Alakzatok;
```

Az Ada83-as szintaxishoz képest az egyedüli változás tehát az, hogy azoknál a típusoknál, amelyeket a későbbiekben további komponensekkel szeretnénk kiterjeszteni, oda kell írni a `tagged` kulcsszót.

Származtatásnál egy típusnak csak egy őse lehet.

A példában szereplő `tagged record ... end record;` helyett írható még: `tagged null record;` is, ez a `tagged record null; end record;` rövidítése.

Az `Alakzat` típus egy jelölt típus. A `Pont` ebből származik, és örökli annak komponenseit (ez most az egyetlen `Szín`) és műveleteit. (`Kirajzol` és `Mi_a_Színe`). A származtatott típusnak van egy rekord-kiterjesztéses része, ami további két komponenst ad hozzá a típushoz (az `X_Kord, Y_Kord` koordinátáit). Célszerűen felüldefiniáljuk az ős típus `Kirajzol` műveletét, és hozzáadunk egy új műveletet is (`Elmozdít`).

A `Kör` típust a `Pont`-ból származtatva itt is felüldefiniáljuk az ősétől örökölt `Kirajzol` műveletet.

A csomag használója, ha deklarál egy `P: Pont;` típusú változót, arra hívhatja az átdefiniált `Kirajzol(P)` műveletet, az új `Elmozdít(P, Z, X)`-t, valamint az `Alakzat` típustól változatlanul örökölt `Mi_a_Színe(P)` műveletet. Ebben az esetben a `Kirajzol(P)` művelet kiválasztása statikusan (fordítási időben) történik.

A kiterjesztést átlátszatlan típussal is alkalmazhatjuk, mind az ős, mind a leszármazott típus lehet átlátszatlan, de a leszármazott csak akkor definiálható, ha az ős szerkezete már ismert, vagyis nem lehetséges az átlátszatlan ős típust még ugyanazon csomag specifikációs részében látható módon kiterjeszteni, ahogy a következő példa is mutatja:

12.2. Példa

```
package Oop2 is
 type A is tagged private; -- átlátszatlan jelölt ős típus
 -- műveletei:
 function Af1(Apar1, Apar2 : A) return Integer;
 function Af2(Apar1, Apar2 : A) return A;
```

```
procedure Af3(Apar1, Apar2 : A);

type Ua is new A with private;
    --átlátszatlan kiterjesztés, megengedett
function Af2(Apar1, Apar2 : Ua) return Ua;
    -- Af2 felüldefiniálása
procedure Af4(Apar : A); -- új művelet

-- látható kiterjesztés nem megengedett,
-- amíg az A teljes típusa nem ismert:
type RosszA is new A with record
   A2:Integer;
end record;

private
   type A is tagged record
      A1:Integer:=1;
   end record;
   type Ua is new A with null record;
end;
```

Megengedett ugyanakkor másik csomagban az eredetileg átlátszatlan típus látható kiterjesztése, például:

12.3. Példa

```
with Oop2; use Oop2;
-- innen jön az átlátszatlan A ős típus

package Oop3 is
--ebben a csomagban már lehet átlátszatlan
-- és látható módon is kiterjeszteni az ős típust

type Uja is new A with private;  -- átlátszatlan kiterjesztés
function Af2(Apar1, Apar2: Uja) return Uja;
type Uja1 is new A with null record; -- látható kiterjesztés
function Af2(Apar1, Apar2: Uja1) return Uja1;

private
   type Uja is new A with record
      A2:Integer:=2;
   end record;
end;
```

A jelölt típusoknak is lehetnek diszkriminánsaik, ezt a leszármazottak is öröklik, ahogy a következő példában látható:

12.4. Példa

```
package Oop4 is
type Méret is range 0..20;
type Atomb is array(Méret range <>) of Integer;
type AD(M: Méret) is tagged private;
...
type UjaD is new AD with private;
...
private
type AD(M: Méret) is tagged record
  A1:Atomb(1..M);
end record;
type UjaD is new AD with null record;
end;
```

A jelölt típus esetén nem megengedett, hogy a diszkrimináns részének legyen alapértelmezett kezdeti értéke, mert ez a nyelv bonyolultságát túlzottan növelné anélkül, hogy tényleges előnyöket jelentene.[3]

12.3. Primitív műveletek

A származtatás során az ún. *primitív művelet*ek öröklődnek. Csak azoknak a típusoknak lehetnek primitív műveletei, amelyeket egy csomag specifikációs részében definiálunk.

A `T` típus primitív műveletei ekkor azok az alprogramok, amelyek közvetlenül a `T` típus definiálása után állnak a specifikációs részben és az alprogramok deklarációjában a paraméterek, illetve az esetleges visszatérési érték típusaként szerepel a `T` típus.[4]

Egy primitív művelet pontosan egy jelölt típusra vonatkozhat, egy származtatott típus definíciója lezárja a szülő primitív műveleteinek deklarációját (ez az ún. „befagyasztó szabály").

Például az `Oop2` csomagban az `A` típusnak primitív műveletei az `Af1`, az `Af2` és az `Af3`, vagyis ezeket örökli az `Uja` típus, és bár művelete, de nem primitív művelete, így nem is öröklődik az `Af4` eljárás, mert annak deklarációja az `Uja` típus deklarációja után szerepel.

[3] A GNAT fordító szintaktikailag elfogadja.

[4] Az Ada95-ben, – ellentétben a legtöbb objektum-orientált programozási nyelvvel, mint például a C++, a Java, vagy az Eiffel, nincs külön szintaktikai elem, amely egybefogná az absztrakt adattípus specifikációját.

Egy öröklött művelet hívása a szülő megfelelő műveletének hívását jelenti, így abban természetesen a leszármazott(ak)ban esetleg definiált új adattagok nem játszanak szerepet.

A műveletekben a jelölt típusú paraméterek mindig referenciaként kerülnek átadásra.

A szintaxis nem írja elő, hogy valamilyen speciális módon (pl. obj.fv(...)) jelöljük az objektumot, aminek a művelet „üzenete".[5]

12.4. Konverzió

A konverziók egyik fajtája az ún. *érték konverzió*. Ekkor az egyes típusok között csak explicit típuskonverzió megengedett. A leszármazott típusú objektum a típus nevének használatával konvertálható az ős típusra, ekkor az objektum kiterjesztett részét nem veszi figyelembe, azokat levágja.

Például az előző `Pont` típusú objektumok konvertálhatóak `Alakzat` típusú objektumra, ekkor az `X_Kord` és `Y_Kord` mezőket levágja, például:

```
O: Alakzat;
P: Pont;
...
O := Alakzat(P);   --megengedett, a P Szín mezője átmásolódik
```

Az Ada95 egyik érdekes és más nyelvekben igen ritkán megvalósított lehetősége az, hogy egy objektumot az ő típusából leszármaztatott típusú objektummá is konvertálhatunk, ha megadjuk a kiterjesztés során definiált új mezők értékét. Például írhatjuk pozícionális hivatkozással:

```
P := (O with 5.0, 13.0);
```

vagy név szerintivel:

```
P := (O with  X_Kord => 5.0,  Y_Kord => 13.0);
```

A konverziók egy másik lehetősége az ún. *nézet konverzió*. Ez lényegében azt jelenti, amit a primitív műveletek öröklésénél leírtunk: ha `T` egy jelölt típus, `ot` egy `T` típusú objektum, s `f` a `T` egy primitív művelete, akkor származtatva `T`-ből `T1`-t, ez örökli `f`-t, s ha `ot1` egy `T1` típusú objektum, akkor `f(ot1)` függvényhívás esetén a hívás idejére `ot1` `T1`-ben definiált adattagjai láthatatlanná válnak (de nem csonkolódnak!).

[5]ellentétben sok más objektum-orientált programozási nyelvvel, mint pl. a C++, Java, Eiffel stb.

12.5. Jelölt típusok a variáns rekord helyett

Nagy rendszerek programozásakor a megbízhatóság támogatásához alapvető elv, hogy a specifikáció módosításakor (ami elkerülhetetlen) a program minél kisebb részét írjuk át.

Tekintsünk egy gyakorlati példát, amely megmutatja az objektum-orientált megközelítés előnyeit a „hagyományos" szemlélettel összehasonlítva. A példát a [3]-ban található leírásra alapoztuk. Ha a rendszerben jelölt típusok használatával típus-hierarchiákat alakítunk ki, akkor úgy adhatunk hozzá újabb funkciókat, hogy a már meglévő, biztonságosan működő részeket a legkisebb mértékben sem kell módosítani. Érdemes összehasonlítani a „hagyományosan" programozott megoldást az objektum-orientált tervezést támogató megoldással.

Tekintsünk egy biztonságvédelmi rendszert, ahol a különböző eseményeket fontosságuknak megfelelő módon kezelik. Az eseményeknek háromféle prioritása legyen:

- **a legalsó szint** – itt ún. eseménynaplózással kigyűjtjük a nyomtatóra az azonnali beavatkozást nem igénylő üzeneteket,
- **a középső szint** – itt az eseménynaplózáson kívül figyelmeztető jelzést is küldünk a kezelőnek, neki ezt „nyugtáznia" kell, s a riasztás csak ekkor szűnik meg,
- **a legfelső szint** – itt az előzőeken túl hangjelzést is adunk, ha egy adott időn belül nem történik „nyugtázás".

A nyomtatón, vagy egy háttértárolón kívül az üzenetet – típusától függően – a terminál-képernyőn, vagy a kivetítő képernyőn is megjelenítjük.

Nézzük először, hogyan oldható meg a feladat a „hagyományos" eszközökkel: A különböző szintű események unióját egy variáns rekord segítségével írhatjuk le. Itt egy elég bonyolult egymásba ágyazott variáns rész tükrözi a szintek logikai összefüggését, nevezetesen, hogy a legfelső prioritású esemény bekövetkezésekor mindazt meg kell tenni, amit a középső szint esetén, s ezen kívül bizonyos időn túl hangjelzést is kell adni. (Lehetne a kódot is többszörözni, ekkor azonban a szerkezet nem tükrözné az események logikai felépítését.)

12.5. Példa

```
with Calendar;
package Biztonságvédelmi_Rendszer is
 type BV_Alkalmazott is
```

```
  type Prioritás is (Alacsony, Közepes, Felsô);
  type Készülék is (Nyomtató, Képernyô, Kivetítô_Képernyô);

  type Esemény (P: Prioritás) is
    record
       Beérkezés_Ideje: Calendar.Time;
       Üzenet: Text;
       case P is
          when Alacsony => null;
          when Közepes | Felsô =>
             Kezelô: BV_Alkalmazott;
                case P is
                   when Alacsony | Közepes => null;
                   when Felsô =>
                          Riasztó_Csengô_Ideje: Calendar.Time;
                 end case;
       end case;
    end record;

  procedure Kijelez(A: in Esemény; On: in Készülék);
  procedure Kezel(A: in out Esemény);

  procedure Naplóz(A: in Esemény);
  procedure Riasztást_Beállít(A: in Esemény);

end Biztonságvédelmi_Rendszer;
```

A Kezel eljárás törzse ekkor a következő lehet:

12.6. Példa

```
procedure Kezel(A: in out Esemény) is
 begin
    A.Beérkezés_Ideje := Calendar.Clock;
    Naplóz(A);
    Kijelez(A, Nyomtató);
    case A.P is
       when Alacsony => null;  -- semmi különös
       when Közepes | Felsô =>
          A.Kezelô := Kezelôt_Kijelöl;
                      --hívja a megfelelö eljárást
          Kijelez(A, Képernyô);
          case A.P is
             when Alacsony | Közepes => null;
             when Felsô =>
```

```
                Kijelez(A, Kivetítô_Képernyô );
                Riasztást_Beállít(A);
          end case;
    end case;
end Kezel;
```

Ennek a megközelítésnek az egyik problémája az, hogy a programkód az egymásba ágyazott variáns rekord szerkezet kezelése miatt elég bonyolult, a komolyabb gondot ugyanakkor az jelenti, hogy, ha mondjuk egy újabb szintű eseménnyel – például egy vészjelzést kiváltó riasztással szeretnénk bővíteni az előzőeket, akkor az egész rendszert módosítani kell. Így a már helyesnek tekinthető kód megbízhatósága is megszűnik.

Az objektum-orientált megközelítésben, amit az Ada95 is támogat, *jelölt* típusokat, és különálló `Kezel` eljárásokat használhatunk az egyes prioritásokhoz. Nincs szükség sem egymásba ágyazott variáns rekordokra, sem egymásba ágyazott `case` utasításokra, sőt voltaképpen a `Prioritás` típusra sem, hiszen ezt ekkor már nem mint a típus egy komponensét fogjuk tárolni.

Így az általános `Esemény` jelölt típusból származtathatjuk az

- `Alacsony_Szintû_Esemény`,
- `Közepes_Szintû_Esemény`,
- `Felsô_Szintû_Esemény`

típusokat, s mindegyiknek megírhatjuk saját speciális `Kezel` műveletét. A csomag specifikáció ekkor:

12.7. Példa

```
 with Calendar;
 package Új_Biztonságvédelmi_Rendszer is
   type Készülék is (Nyomtató, Képernyô, Kivetítô_Képernyô);
   type BV_Alkalmazott is
...
   type Esemény is tagged
     record
         Beérkezés_Ideje: Calendar.Time;
         Üzenet: Text;
    end record;

 -- Az Esemény típus primitív mûveletei:
   procedure Kijelez(A: in Esemény; On: in Készülék);
   procedure Kezel(A: in out Esemény);
   procedure Naplóz(A: in Esemény);
```

```
  type Alacsony_Szintû_Esemény is new Esemény with
    null record;
-- Az Alacsony_Szintû_Esemény típus nem módosítja az
-- örökölt primitív mûveleteket

  type Közepes_Szintû_Esemény is new Esemény with
    record
       Kezelô: BV_Alkalmazott;
    end record;

-- a Közepes_Szintû_Esemény típus specializálja
-- az örökölt Kezel mûveletet
  procedure Kezel(MA: in out Közepes_Szintû_Esemény);

  type Felsô_Szintû_Esemény is new Közepes_Szintû_Esemény
    with record
       Riasztó_Csengô_Ideje: Calendar.Time;
  end record;

-- a Felsô_Szintû_Esemény típus specializálja
-- az örökölt Kezel mûveletet
-- és bevezeti az új Riasztást_Beállít mûveletet
  procedure Kezel(HA: in out Felsô_Szintû_Esemény);
  procedure Riasztást_Beállít(HA: in Felsô_Szintû_Esemény);

end Új_Biztonságvédelmi_Rendszer;
```

Megjegyezzük, hogy az Ada95 megengedi, hogy egy származtatott típus ugyanabban a csomag specifikációban szerepeljen, mint az ős típus, és örökölje annak primitív műveleteit. A szülőhöz azonban nem lehet új primitív műveletet hozzáadni azután, hogy egy típust származtattunk belőle (ti. a befagyasztó szabály miatt).

Az `Alacsony_Szintû_Esemény` típus egyszerűen az `Esemény` típus egy másolata (ezt a `with null record;` fejezi ki), s örökli annak primitív műveleteit.

A `Közepes_Szintû_Esemény` típus kiterjeszti az `Esemény` típust, bevezeti az új `Kezelô: BV_Alkalmazott;` komponenst, és specifikálja a megfelelő `Kezel` eljárást.

A `Felsô_Szintû_Esemény` típus kiterjeszti a `Közepes_Szintû_Esemény` típust, és specifikálja saját `Kezel` műveletét. Így egy bonyolult `case` utasítást tartalmazó `Kezel` eljárás helyett az Ada95 objektum-orientált megoldása szerint redundancia nélkül szétoszthatjuk a teendőket az események kezelésének logikája alapján minden speciális `Esemény` típusra.

Az Alacsony_Szintû_Esemény típus, a Közepes_Szintû_Esemény típus, és a Felsô_Szintû_Esemény típus változtatás nélkül öröklik a Kijelez és Naplóz műveleteket. Végül a Felsô_Szintû_Esemény típus bevezeti a Riasztást_Beállít műveletet, amely az alacsonyabb szintű eseményeknél még nem szükséges.

A csomag törzse ekkor a következő lehet:

12.8. Példa

```
package body Új_Biztonságvédelmi_Rendszer is
  procedure Kezel(A: in out Esemény) is
    begin
      A.Beérkezés_Ideje := Calendar.Clock;
      Naplóz(A);
      Kijelez(A, Nyomtató);
  end Kezel;

  procedure Kezel(MA: in out Közepes_Szintû_Esemény) is
    begin
      Kezel(Esemény(MA)); -- hívja az Esemény östípus
                          -- megfelelö müveletét
      MA.Kezelô := Kezelôt_Kijelöl;
                          -- hívja a megfelelö eljárást
      Kijelez(MA, Képernyô);  -- és kijelez a képernyöre is
  end Kezel;

  procedure Kezel(HA: in out Felsô_Szintû_Esemény) is
    begin
      Kezel(Közepes_Szintû_Esemény(HA));
                        -- kezeli mindazt, amit az öse
      Kijelez(HA, Kivetítô_Képernyô );
                        -- jelez a Kivetítô_Képernyô-n is
      Riasztást_Beállít(HA); --vészcsengöt állít, ha kell
  end Kezel;
  procedure Kijelez(A: in Esemény; On: in Készülék) is
     separate;
  procedure Naplóz(A: in Esemény) is
     separate;
  procedure Riasztást_Beállít(HA: in Felsô_Szintû_Esemény) is
     separate;

end Új_Biztonságvédelmi_Rendszer;
```

A Kezel művelet minden törzse pontosan csak annyi kódot tartalmaz, ami a megfelelő altípusnál szükséges, és explicit típuskonverzió használatával

hívja az őse megfelelő műveletét. Itt minden típusellenőrzés még statikus, fordítási időben történik.

Ebben a modellben egy új eseményfajta – a `Vész_Riasztás_Esemény` például – bevezetése nem igényli a már ellenőrzött kód módosítását – ennek következtében újratesztelését sem, egyszerűen egy új csomagban definiáljuk az `Esemény` típus egy újabb leszármazottját:

12.9. Példa

```
with Új_Biztonságvédelmi_Rendszer;
package Vész_Riasztás_Kezelô_Rendszer is

 type Vész_Riasztás_Esemény is
  new Új_Biztonságvédelmi_Rendszer.Esemény with private;
  -- a kód tényleges megvalósítását elrejthetjük
  -- a felhasználók elöl

 procedure Kezel(EA: in out Vész_Riasztás_Esemény);
   -- átdefiniáljuk megfelelöen a kezelö müveletet

   -- söt most a Kijelez és a Naplóz müveleteket is:
 procedure Kijelez(EA: in Vész_Riasztás_Esemény;
              On: in Új_Biztonságvédelmi_Rendszer.Készülék);

 procedure Naplóz(EA: in Vész_Riasztás_Esemény);

private
       ... -- leírjuk a választott reprezentációt
end Vész_Riasztás_Kezelô_Rendszer;
```

Az eredeti „hagyományos" megvalósításban a variáns rekord típust át kellett volna definiálni, s így minden ezt használó programkódot megfelelően módosítani kellett volna – s csak jelentős többlet-munkaráfordítással lehetett volna elérni a rendszer eredeti megbízhatósági szintjét.

12.6. Többalakúság

12.6.1. Osztályok és futásidejű azonosítás

Az eddig látott lehetőségek segítségével módunk nyílt már meglévő típusra alapozva, annak kiterjesztéseként új típusok definiálására. Így létrehoztuk az események hierarchiáját, önálló típusként kezelve ezeket. Az objektumok aktuálisan végrehajtandó műveleteit fordítási időben határoztuk meg. Ezt neveztük *statikus összekapcsolás*nak.

Szükség van még olyan eszközre is, melynek segítségével egy kialakított típus-hierarchia bármely típusába tartozó objektumot akkor is speciális tulajdonságainak megfelelően tudunk kezelni, ha az objektum aktuális típusa csak futási időben ismert – ekkor *dinamikus összekapcsolás*sal választjuk ki a megfelelő műveletet.

Az Ada az ún. *osztályszintű típus* fogalmát vezeti be, hogy lehetővé tegye egy `T` jelölt típus és az összes belőle származtatott típus *polimorfikus* kezelését. Minden `T` jelölt típushoz létezik egy `T'Class` típus, mely tartalmazza mindazon típusok *unióját*, melyek a `T` gyökerű származtatási fában vannak. A `T'Class` típus értékei tehát a `T` típus értékei és az összes belőle származtatott típus értékei.

Így az `Esemény'Class` típus típusértékhalmaza tartalmazza az `Esemény` típus, és az összes lehetséges belőle származtatott típus értékeit, így az `Esemény`, az `Alacsony_Szintû_Esemény`, a `Közepes_Szintû_Esemény`, valamint a `Felsô_Szintû_Esemény` típusok értékhalmazát, sőt az esetleg később definiáltakat is.

A `T`-ből származtatott bármely típus egy értéke implicit módon konvertálható `T'Class` típusúvá. A `T'Class` típus megszorítás nélküli típusként viselkedik, ha ilyen típusú változót deklarálunk, annak kötelező kezdeti értéket adni (különben a fordítóprogram nem tudná, mennyi helyet kell neki lefoglalni), s ennek típusa később nem változtatható.

Például írhatjuk:

```
E: Esemény;
Ae: Alacsony_Szintû_Esemény;
Ke: Közepes_Szintû_Esemény;
E1: Esemény'Class:=Ae;
```

de nem írhatjuk:

```
E2: Esemény'Class;
```

Az osztályszintű típushoz definiálhatunk mutató típust is, ennek segítségével kezelhetjük a heterogén szerkezeteket. Például:

12.10. Példa

```
type Esemény_Mut is access all Esemény'Class;
-- definiálhatjuk az  Esemény osztályra mutató típust

Emut1,Emut2: Esemény_Mut; -- deklarálhatunk ilyen típusú
  -- változókat
....
begin
  Emut1:= new Közepes_Szintû_Esemény;
  Emut2:=Emut1;
```

```
   Emut1:=new Alacsony_Szintû_Esemény;
                       -- átállíthatjuk a mutatót az
                       -- Esemény'Class tetszöleges típusára
```

Az osztályszintű változóknál a típusok közti megkülönböztetést az objektumok *"címke"* ('tag') része teszi lehetővé. Ez a 'Tag attribútummal le is kérdezhető:

```
 E1: Esemény'Class:=...;
 E2: Esemény'Class:=...;
 begin
 if E1'Tag=E2'Tag then
   ...
 end if;
 ....
 end;
```

Az osztályszintű típusok objektumait tehát úgy tekinthetjük, mint egy párt, aminek első tagja egy címke, ami megadja az osztályon belüli típust; második tagja egy érték, ami a címkének megfelelő típusérték. Ezek a (címke, érték) párok erősen típusosak, az Ada filozófiájának megfelelően.

A következő példában az osztályszintű változó kezdeti értékét futás közben állítjuk be. Az átlátszatlan `Fájl` típus primitív művelete, amelyet a leszármazottakban megfelelően átdefiniálhatunk, `Megtekint`, és a `Fájl'Class` típusúnak deklarált változó a felhasználó által megadott kezdeti értékének megfelelő `Megtekint` műveletet választja ki:

12.11. Példa

```
type Fájl is tagged private;
procedure Megtekint(F: Fájl);
......
type Direktori is new Fájl with private;
procedure Megtekint(D: Direktori);
   -- listázza az alkönyvtárat
......
type Ada_Fájl is new Fájl with private;
procedure Megtekint(A: Ada_Fájl);
   -- más színnel mutatja a kulcsszavakat
type Ada_Könyvtár is
      new Direktori with private;
procedure Megtekint(L: Ada_Könyvtár);
  --listázza az L könyvtári egységeit és állapotukat
......
 declare
```

```
   Egy_Fájl: Fájl'Class := Kér_a_Felhasználótól;
   begin
      Megtekint(Egy_Fájl);
       -- a megfelelô megtekintö müveletet hívja
       --  dinamikusan, futás közben
 end;
```

Az osztályszintű típusoknak nincsenek saját primitív műveleteik.

Műveleteket természetesen tudunk definiálni osztályszintű típusú argumentummal. Ezek a műveletek az osztályszintű műveletek, és alkalmazhatóak minden, az osztályba tartozó típus objektumára.[6]

Például írhatunk az előző `Fájl'Class` típusnak műveleteket:

```
function Méret_Bájtokban(Egy_Fájl: Fájl'Class) return Natural;
function Kér_a_Felhasználótól return Fájl'Class;
```

Hasonlóképpen definiálhatunk egy, az események sorozatát kezelő alprogramot, mely `Esemény'Class`-beli argumentumot vár, s ezt nem is szűkíthetjük le, ha meg akarjuk őrizni annak lehetőségét, hogy a rendszert később kibővítsük:

```
procedure Eseményeket_Kezel(AC: in out Esemény'Class) is
....
begin
...
Kezel(AC);
...
end;
```

Ekkor az `Eseményeket_Kezel` eljárás aktuális paramétere tetszőleges, `Esemény` típushoz, vagy annak valamelyik leszármazottjához tartozó érték lehet, ezt (címke, érték) párként kezeli, s a törzsben a címke által meghatározott, az aktuális típusnak megfelelő `Kezel` műveletet hívja majd meg.

12.6.2. Kiválasztás

Egy jelölt típus egy primitív műveletét meghívhatjuk egy osztályszintű paraméterrel is, a végrehajtandó műveletet ekkor az objektum címkéjének megfelelő aktuális típussal futási időben választja ki.

A futás idejű kiválasztásban csak a primitív műveletek vesznek részt, hiszen csak ezek öröklődnek.

[6] Az ilyen alprogramok sok szempontból hasonlóak a korlátozás nélküli tömb típusú argumentummal rendelkező alprogramokhoz.

A fenti `Fájl`-os példában a `Megtekint` primitív műveletet a leszármazottak felüldefiniálják, így ez részt vesz a futás idejű kiválasztásban.

Ha a `Megtekint` műveletet egy `Fájl'Class` típusú osztályszintű paraméterrel hívjuk meg, akkor az aktuális paraméter címke része fogja eldönteni (futási időben), hogy melyik `Megtekint`-nek kell végrehajtódnia.

Másrészt, ha a `Megtekint` műveletet egy konkrét, például `Fájl`, vagy `Direktori` stb. típusú aktuális paraméterrel hívnánk meg, akkor már fordítási időben eldőlne, hogy melyik `Megtekint` műveletet fogja végrehajtani.

Hasonlóképpen lehetséges például, hogy a biztonságvédelmi rendszerben az egyes eseményeket feldolgozás előtt egy heterogén, `Esemény'Class`-beli elemeket tartalmazó sorban tároljuk, a

```
type Esemény_Mut is access Esemény'Class;
```

mutató típus segítségével, s ekkor, ha a `Kov_E: Esemény_Mut;` változóban tároljuk a következő esemény mutatóját, akkor a

```
Kezel(Kov_E.all);
```

hívás hatására is dinamikusan, futási időben kerül kiválasztásra a megfelelő `Kezel` művelet.

Tehát az, hogy egy adott típushierarchiában, egy primitív művelet speciálisan átdefiniált változatai közül aktuálisan melyik kerül végrehajtásra, eldőlhet fordítási időben (statikus kötés) is, és futási időben is (dinamikus kötés, futás idejű kiválasztás). Ez a művelet meghívásának módjától függ.[7]

Lehetséges paraméter módok:

Aktuális/Formális	Konkrét típus	Osztály szintű típus
Konkrét típus	Statikus kötés	Osztály szintű művelet
Osztály szintű típus	Futás idejű kiválasztás	Osztály szintű művelet

Az osztályszintű műveleteknél, vagyis amikor a formális paraméter típusa osztályszintű, akkor az aktuális paraméter típusától függetlenül mindig ugyanaz a művelet hajtódik végre, így nincs szükség futás-idejű kiválasztásra.

Abban az esetben, amikor mind a formális paraméter, mind az aktuális paraméter típusa konkrét típus, akkor voltaképpen egyszerűen a művelet átlapolásáról van szó, így szintén fordítási időben eldönthető, hogy melyik műveletet kell végrehajtani.

Futás idejű kiválasztásra csak akkor van szükség, amikor a formális paraméter típusa egy konkrét típus, s az aktuális paraméter osztályszintű típusú.

[7]ellentétben pl. a C++-szal, ahol a dinamikus kötést a művelet deklarációjánál kell előírni a `virtual` kulcsszóval

12.6.3. Egy lehetséges megvalósítás

A jelölt típusok tehát azok, amelyek újabb komponensekkel és műveletekkel kiterjeszthetőek. Az ilyen típusú objektumoknak van egy *címke-része* (*tag*), s ez tartalmaz információt az objektum statikus típusáról. A típusok leírója tartalmaz egy táblázatot a típus primitív műveleteinek címeiről.

A futás idejű kiválasztás egy lehetséges megvalósításában a kiválasztás egyszerűen úgy történhet, hogy az aktuális paraméter címkéje alapján először kiválasztja a megfelelő típust, s ennek a primitív műveletei közül keresi ki, s hajtja végre a hívott műveletet (indirekt ugrással). A primitív művelet neve már fordítási időben ismert, csak a konkrétan végrehajtandó művelet címe dől el futás közben, dinamikusan.

12.6.4. Az objektumok címkéje és a 'Tag attribútum

Egy `T` jelölt típus egy objektumának címkéje `T`-t azonosítja, egy `T'Class` osztályszintű típusú objektum címkéje a kezdeti értékadásban kapott kifejezés címkéjével egyezik meg, illetve ha ez egy függvény visszatérési értéke, akkor a visszaadott kifejezés címkéje. Paraméterátadáskor a címke megőrződik.

A `T'Tag` `Tag` típusú attribútum adja vissza ennek a címkének az értékét. Ezt a típust az előre definiált `Ada.Tags` csomag tartalmazza.

12.6.5. Egyenlőség és tartalmazás operátorok

Az előre definiált egyenlőség és tartalmazás műveletek vizsgálatát általánosították az osztályszintű változókra. A műveletek eredménye a címke résztől függ, de ha ezek nem egyeznek meg, vagyis a két osztályszintű változó különböző típusú értéket tartalmaz, akkor az nem fog `Constraint_Error` hibát okozni. A címkék különbségét, mint a nem-egyenlőséget definiálták. Ez lehetővé teszi, hogy két osztályszintű objektumot biztonságosan összehasonlítsunk.

A tartalmazás teszt (`X in S`) szemantikája a következő:

- ha az `X` egyszerű kifejezés típusa osztályszintű, és az `S` típus is egy osztályszintű típus, akkor igazat ad, ha az `X` `T` típusának az `S` őse,
- ha az `X` egyszerű kifejezés típusa osztályszintű, és az `S` típus nem osztályszintű, akkor igazat ad, ha az `X` `T` típusa aktuálisan megegyezik az `S` típussal;
- ha az `X` egyszerű kifejezés típusa nem osztályszintű, és az `S` típus egy osztályszintű típus, akkor igazat ad, ha az `X` `T` típusát az `S` tartalmazza,

- ha az `X` egyszerű kifejezés típusa nem osztályszintű, és az `S` típus sem osztályszintű, akkor igazat ad, ha az `X` `T` típusa aktuálisan megegyezik az `S` típussal.

Így például

```
type A is tagged record
....
end record;
   procedure ....
type B is new A with record
...
end record;
```

esetén, ha a következő deklarációk érvényesek:

```
   Ao: A;
   Bo: B;
   Ac1: A'Class:=Ao;
   Ac2: B'Class:=Bo;
   Ac3: A'Class:=Bo;
```

akkor a tartalmazási relációkról a következőket mondhatjuk: igaz az `Ac1 in A'Class`, illetve az `Ac3 in A'Class` – természetesen, hiszen ilyennek deklaráltuk, de hasonlóképpen igaz az `Ac1 in A` is, mert aktuális értéke `A`-beli, és nem igaz az `Ac3 in A`, mert ennek aktuális értéke `B`-beli, igaz az `Ac2 in A'Class`, mert `B'Class` részhalmaza az `A'Class`-nak, s az aktuális érték miatt igaz az `Ac2 in B`, az `Ac3 in B'Class`, illetve az `Ac3 in B` is, igaz a `bo in A'Class`, de fordítási hibát jelent, ha `bo in A`-t szeretnénk kérdezni, hiszen ezekről már fordításkor eldönthető, hogy nem kompatíbilis típusok.

A tartalmazás teszt lehetőséget ad arra is, hogy futási időben specializáljuk a kódot a dinamikus típustól függően. Ezzel a vizsgálattal legyünk óvatosak, mert ha a típus-hierarchiát újabb leszármazottakkal akarjuk bővíteni, akkor valószínűleg ezeket a kódrészleteket is módosítani kell. Például:

12.12. Példa

```
   procedure Szerkeszt(F_Fájl: in Fájl'Class) is
   ...
      if F_Fájl in Direktori'Class then
-- akkor hajtódik végre, ha F_Fájl típusa Direktori
-- vagy ennek valamelyik leszármazottja
-- pl. Ada_Könyvtár
      ...
      else if F_Fájl in Ada_Fájl then
      -- csak akkor hajtódik végre, ha F_Fájl Ada_Fájl típusú
      ...
```

A második if-ben mindenképpen érdemesebb lenne ellenőrizni, hogy igaz-e az F_Fájl in Ada_Fájl'Class, ugyanis ebben az esetben akkor is ez az ág fog lefutni, ha F_Fájl típusa leszármazottja az Ada_Fájl-nak. Ezt a technikát kell akkor is alkalmazni, hogy ha egy olyan primitív műveletet akarunk meghívni, amely az adott öröklődési hierarchia egy részében van csak definiálva. Ekkor a művelet meghívása előtt le kell ellenőrizni, hogy az osztályszintű objektum aktuális típusa az adott részbe esik-e.[8]

12.6.6. Eltakart implementáció hívása

Ahogy a riasztási rendszerünkben is láthattuk, ha egy típus leszármazottjában felüldefiniáljuk valamelyik műveletet, akkor néha szükség lehet arra, hogy a szülő (eltakart) műveletét hívjuk meg.

12.13. Példa

```
procedure Kezel(HA: in out Felsô_Szintû_Esemény) is
    begin
      Kezel(Közepes_Szintû_Esemény(HA));
  -- kezeli mindazt, amit az ôse
      Kijelez(HA, Kivetítô_Képernyô );
  -- jelez a Kivetítô_Képernyôn is
...stb.
```

Ennek kiválasztásához egyszerűen a típusnevet, mint konverziós függvényt használjuk. Ekkor egy *nézet konverzió* történik, s az ős Kezel műveletete hívásának idejére a paramétert Közepes_Szintû_Esemény típusúként tekinti.

12.6.7. Újrakiválasztás

Bővítsük ki a korábban definiált Alakzat típus primitív műveleteinek halmazát a

```
  procedure Letöröl (O: in Alakzat);
```

művelettel, – ezt is nyilván át kell majd definiálni a leszármazottaknál a Kirajzol művelethez hasonlóan. Ekkor a Pont típus Elmozdít(P: in Pont; X1,Y1:Float); műveletének törzse, (amely áthelyezi az alakzatot), először a Letöröl műveletet hívja, majd az aktuális paraméternek megfelelő új helyen hívja a Kirajzol műveletet:

[8] Érdemes megjegyezni, hogy sablonokban (generic) lokális jelölt típus esetén a nyelv nem biztosítja, hogy az egyes példányokban más lesz a típusok címkéje, globális jelölt típusokra azonban ez biztosan teljesülni fog.

12.14. Példa

```
procedure Elmozdít(P : in Pont; X1,Y1:Float)  is
begin
Letöröl(P);
....-- koordinátát változtat
Kirajzol(P);
end;  --áthelyezi a pontot
```

A `Kör` típus örökli az `Elmozdít` műveletet, de ha azt a fenti formájában meghívjuk, például a

```
...
 k: Kör;
 begin
....
 Elmozdít(k,1.0,1.0);
```

paranccsal, azt tapasztaljuk, hogy hiába definiáltuk felül a `Pont` típustól örökölt `Letöröl` és `Kirajzol` műveleteket, az `Elmozdít` mégis a `Pont` típushoz definiáltakat hívja. (Statikus kötéssel, fordítási időben eldöntve a hívott változatot.) Egy lehetséges megoldás, hogy úgy döntünk, újradefiniáljuk a `Kör` típusnál az `Elmozdít` műveletet is,

```
 procedure Elmozdít(P: in Kör; X1,Y1:Float);
```

alakban, ahol a törzsön jelen esetben nem változtatunk semmit.

Elvben pontosan ilyen az a művelet is, amit a `Kör` típus a `Pont`-tól örököl, így az átdefiniálást valójában teljesen fölöslegesnek érezzük, mégis, így – megint csak statikus kötéssel ugyan – az előző `Elmozdít(k,1.0,1.0);` hívás most jól működik, a `Kör` típus `Letöröl` és `Kirajzol` műveleteit hívja. Ha azt szeretnénk, hogy ilyen és hasonló helyzetekben egy adott típus leszármazottainál a változatlan formában „átdefiniált" műveletet átdefiniálás nélkül is úgy használhassuk hogy az törzsében már a leszármazott típus(ok) megfelelően átdefiniált műveletét – dinamikus kötéssel, futási időben – válassza ki, akkor a törzsben azon műveletek paraméterének típusát, amelyeket dinamikus kiválasztással szeretnénk hívni, a szülő osztályszintű típusává kell konvertálni.

Például, ha a `Pont` típus `Elmozdít` műveletének a törzse:

12.15. Példa

```
procedure Elmozdít(P : in Pont; X1,Y1:Float)  is
begin
  Letöröl (Pont'Class(P));
  ... --koordinátát változtat
  Kirajzol(Pont'Class(P));
end;  --áthelyezi a pontot
```

akkor az előző

```
Elmozdít(k,1.0,1.0);
```

hívás már jól működik, s az `Elmozdít` műveletet csak akkor kell átdefiniálni valamelyik leszármazottnál, ha ténylegesen a törzs kódján is akarunk változtatni. Ezt a technikát hívják az Adában *újrakiválasztásnak*.

Hasonlóképpen például tekintsük az előző `Fájl` típus-hierarchiát, s legyen a `Kódol` a `Fájl` jelölt típus egy új primitív művelete, amely a fájl típusától függetlenül kódolja azt, majd megjeleníti, de már az aktuális típusától függően. Ekkor a `Kódol`-t örökli a `Fájl` típus minden leszármazottja, de ahhoz, hogy meghívásakor az aktuális paraméter típusának megfelelő `Megtekint` műveletet hívja, a paraméter típusát `Fájl'Class`- típusúként kell kezelni.

```
...
   procedure Kódol(F : Fájl) is
   begin
      ...
      Megtekint(Fájl'Class(F)); -- újrakiválasztás
   end;
```

12.7. Absztrakt típusok és műveletek

Az öröklődési hierarchia kialakításában gyakran hasznosak lehetnek azok a típusok, (pontosabban típusspecifikációk) amelyek a származtatás során kialakított típusok közös alapját képezik. Például ilyen valójában az előzőekben felvázolt `Geometriai_Alakzatok` csomagban az `Alakzat` típus, vagy a biztonságvédelmi rendszer `Esemény` típusa.

Ezek elvileg fontosak, egybefogják a különböző konkrét leszármazottakat. Az Ada95 lehetőséget ad arra, hogy típusspecifikációkat deklaráljunk, anélkül, hogy megadnánk a konkrét reprezentációt, illetve a művelet(ek) implementációját, majd a származtatás során határozzuk meg az egyes leszármazottak reprezentációját, illetve a művelet(ek) speciális implementációját. Ehhez a jelölt típusokat és primitív műveleteiket `abstract`-nak deklarálhatjuk.

Például írhatjuk:

```
type A is abstract tagged null record;
```

vagy

```
procedure f(Apar: A) is abstract;
```

Az absztrakt műveletek egyik legfontosabb jellemzője, hogy nincs törzs részük, azaz az implementáció hiányzik. Ennek következménye, hogy konkrét típus származtatásánál az absztrakt műveleteknek kötelező a törzset megadni.

Ugyanakkor az is igaz, hogy egy absztrakt típusnak is lehetnek konkrét műveletei, s ezeket nem kötelező felüldefiniálni.

Absztrakt típusból származtathatunk újabb, akár absztrakt, akár konkrét típust.

Az absztrakt típusok osztályába tartozó változókra a műveleteket ezután dinamikus kiválasztással hívhatjuk. Egy absztrakt típusnak önmagában természetesen nem lehet objektuma.

Az absztrakt típusokból is építhetünk hierarchiát. Amikor már minden primitív műveletet helyettesítettünk valódi (implementációt is tartalmazó) művelettel, akkor a típust már nem kötelező absztraktként deklarálni, így objektuma is lehet, a szokásos módon.

Az említett `Geometriai_Alakzatok` csomagban például célszerűbb az `Alakzat` típust absztraktnak deklarálni:

12.16. Példa

```
package Geometriai_Alakzatok2 is
  type Alakzat is abstract tagged record
    Szín: Képernyô_Szín := Fehér;
   end record;
  -- Az Alakzat típus primitív müveletei:
  -- még nincs törzse:
  procedure Kirajzol(O: in Alakzat) is abstract;
  -- lehet  olyan müvelet is, aminek van törzse:
  function Mi_a_Színe(O: in Alakzat) return Képernyô_Szín;
  -- a konkrét származtatás ugyanúgy  lehetséges:
  type Pont is new Alakzat with record
  ...
  end record;
  procedure Kirajzol(O: in Pont);
  -- kötelezö az implementálása
   ....
  type Kör is new Pont with record
   ....
  end record;
  procedure Kirajzol(O: in Kör);
  -- nem kötelezö az implementálása, de ajánlott
   ....

end Geometriai_Alakzatok2;
```

Hasonlóképpen a biztonságvédelmi rendszert is átfogalmazhatjuk úgy, hogy az `Esemény` gyökér típus absztrakt legyen, sőt bevezethetjük a tetszőleges, az `Esemény'Class` típus-hierarchiában definiált típusú eseményre mutató `Esemény_Mut` típust is. Mindezt deklarálhatjuk egy különálló, törzs nélküli csomagban:

12.17. Példa

```
package BV_Rendszer_Alap is
  type Esemény is abstract tagged null record;
  type Esemény_Mut is access all Esemény'Class;
  procedure Kezel(A: in out Esemény) is abstract;
end BV_Rendszer_Alap;
```

Majd később hozzáadhatjuk a normál biztonságvédelmi rendszer különböző prioritású eseményeit:

12.18. Példa

```
with BV_Rendszer_Alap, Calendar;
package Normál_Biztonságvédelmi_Rendszer is
  type Készülék is (Nyomtató, Képernyô, Kivetítô_Képernyô );
  type BV_Alkalmazott is
       ....
  type Alacsony_Szintû_Esemény is new
       BV_Rendszer_Alap.Esemény with
    record
      Beérkezés_Ideje: Calendar.Time;
      Üzenet: Text;
   end record;

-- Az Alacsony_Szintû_Esemény típus primitív mûveletei azok,
-- amelyek korábban az Esemény típus mûveletei voltak:
  procedure Kezel(A: in out Alacsony_Szintû_Esemény);
  procedure Kijelez(A: in Alacsony_Szintû_Esemény;
                    On: in Készülék);
  procedure Naplóz(A: in Alacsony_Szintû_Esemény);
--
  type Közepes_Szintû_Esemény is new
       Alacsony_Szintû_Esemény with
     record
     ....
  end record;
  ....
end Normál_Biztonságvédelmi_Rendszer;
```

Ebben az új megközelítésben az `Alacsony_Szintû_Esemény` típushoz adjuk meg az első reprezentációt, s a konkrét műveleti törzset.

A `Közepes_Szintû_Esemény` típust ezek után természetes módon származtathatjuk az `Alacsony_Szintû_Esemény`-től.

Külön megfontolást igényel, hogy mely műveleteket deklaráljunk a legfelsőbb, absztrakt szinten?

Erre a kérdésre a választ alapvetően a külső környezet igényei határozzák meg, így például az `Eseményeket_Kezel` globális célú eljárás alapvetően csak a `Kezel` műveletet használja, nem tűnik szükségesnek, hogy a `Kijelez` és a `Naplóz` műveletek is absztraktként deklarálva legyenek.

A megfelelő változtatásokat a csomag törzsében is végre kell hajtani, például az `Alacsony_Szintû_Esemény` típus `Kezel` műveletének törzsében kell megadni azt, amit eddig az `Esemény` típusnál írtunk le, s mivel változott az öröklődési hierarchia, így a `Közepes_Szintû_Esemény` átdefiniált `Kezel` művelete az új szülő-típus, azaz az `Alacsony_Szintû_Esemény` típus `Kezel` műveletének hívásával kezdődik:

12.19. Példa

```
package body Normál_Biztonságvédelmi_Rendszer is

  procedure Kezel(A: in out Alacsony_Szintû_Esemény) is
    begin
      A.Beérkezés_Ideje := Calendar.Clock;
      Naplóz(A);
      Kijelez(A, Nyomtató);
  end Kezel;

  procedure Kezel(MA: in out Közepes_Szintû_Esemény) is
    begin
      Kezel(Alacsony_Szintû_Esemény(MA));
            -- hívja az Alacsony_Szintû_Esemény
            -- östípus megfelelö müveletét
            -- stb., mint korábban
      .....
```

Ha ezek után egy új eseményfajtát - például a `Vész_Riasztás_Esemény` típust akarunk bevezetni, akkor eldönthetjük, hogy ezt az absztrakt `Esemény` típusból, vagy valamilyen okból a már meglévő egyéb leszármazottakból tesszük-e.

Az absztrakt típusok használatának egy lehetséges példája az, amikor egy adott típusspecifikáció többféle lehetséges reprezentációját a konkrét leszármazottakban adjuk meg, s a közös, általános jellemzőket foglaljuk össze az absztrakt adattípus szintjén, mint például a sor, verem stb. jellegű adattípusoknál:

12.20. Példa

```
generic
   type Elem is private;
package Verem_Osztály is
   type Verem is abstract tagged limited private;
   procedure Betesz(S: in out Verem; I: in  Elem)
                                             is abstract;
   procedure Kivesz(S: in out Verem; I: out Elem)
                                             is abstract;
   function  Méret(S: Verem) return Natural;
   -- ez a művelet nem absztrakt!
private
  type Verem is abstract tagged limited
    record
      Hossz: Natural;
  end record;
end Verem_Osztály;
```

A `Méret` művelet nem absztrakt, így kell hozzá implementációt is adni, de ha származtatunk az absztrakt `Verem` típusból, s megadjuk teljes reprezentációját, akkor ott nem kötelező felüldefiniálni ezt a műveletet.

12.8. Konstruktorok, destruktorok, értékadás

Az absztrakt adattípusok megvalósításánál fontos kérdés, hogy az objektumok létrehozásakor a rendszer állítsa be a típusinvariánst, vagyis állítsa be az objektum kezdeti állapotát a típusspecifikációban előírtnak megfelelőre.

Ugyanígy, az objektumok felszabadítása sem triviális feladat – gondoljunk például egy láncolt listával ábrázolt veremre.

Az ilyen feladatokat ellátó műveleteket hívják az objektum konstruktorának, illetve destruktorának.

Ada95-ben is lehetőség van konstruktorok és destruktorok definiálására, bár a más nyelveknél megszokottól ez is egy kicsit eltér. Ahogy a 7. fejezetben szerepel, itt létezik két előredefiniált jelölt típus, a `Controlled` és a `Limited_Controlled` s ha ezek leszármazottaiban az ős adott primitív műveleteit (`Initialize`, illetve `Finalize`) felüldefiniáljuk, megadhatjuk, hogy az adott típusú objektum létrehozásakor, illetve felszabadításakor milyen rutin fusson le.

A `Controlled` típusból származtatott típusú objektumoknál az `Adjust` művelet az értékadás utolsó lépéseként aktivizálódik, miután a memóriaterület byte-ról byte-ra történő átmásolása befejeződött.

A `Limited_Controlled` típusból származtatott típusok objektumaira nem értelmezett az értékadás művelete.

A `Finalization` csomag specifikációjából látszik, hogy minden jelölt típusnak csak egy konstruktora lehet, és csak azonos típusok között értelmezett az értékadás.

12.9. Feladatok

12.1. Valósítsa meg átlátszatlan jelölt rekordok segítségével a következő típushierarchiát:

A `Helyiség`-eket egy egész szám azonosítja, egy sztringben megadhatunk egy leírást róluk, s tudnunk kell az alapterületüket is – ez egy valós szám.

A `Raktár`, a számítógépes `Labor` és az `Iroda` speciális `Helyiség`-fajták, az `Iroda` `Munkaszoba` vagy `Tárgyaló` lehet.

A `Raktár`-ak kapacitását létrehozáskor adjuk meg: a `Termékkód`-ot és a `Max_Darabszám`-ot tartalmazó párok fix hosszúságú sorozataként. Tároljuk az egyes termékek aktuális darabszámát is, valamint a raktáros nevét.

A `Labor`-okról tudjuk, hogy hány számítógépes munkahelyet tartalmaznak, a `Munkaszoba`-ról, hogy 1, 2 vagy 3 személyesek, és ki(k) dolgoznak bennük - ezek az adatok változhatnak. A `Tárgyaló`-ról tároljuk a maximális és az aktuális befogadóképességét.

A primitív műveletek tegyék lehetővé az információk megfelelő kezelését.

12.2. Valósítsa meg átlátszatlan jelölt rekordok segítségével a következő típushierarchiát:

Az `Alkalmazott` típus tartalmazza az alkalmazottak nevét és azonosítóját - mindkettő sztring típusú, valamint fizetését - ez egy egész szám.

A `Fônök` és a `Titkárnô` típusok az `Alkalmazott` típusból származtatással hozhatóak létre.

A `Fônök` típus tartalmazza a beosztottak számát, a `Titkárnô` típus az életkort.

Az `Igazgató` típus a `Fônök` típus leszármazottja.

A primitív műveletek tegyék lehetővé az információk megfelelő kezelését. Definiáljon egy `Éves_Szabadság` függvényt is, amely más-más algoritmust használ az egyes típusok esetén!

12.3. Bővítse ki az `Alakzat` osztályt további síkidomokkal, és valósítsa meg a `Kirajzol` műveleteket!

13. Fejezet

Konténer típusok

Egy *konténer*ben adott típusú objektumok tárolhatóak, majd később megtalálhatóak, tetszés szerint visszakereshetőek. A tömbök, listák, vermek, sorok stb. mind-mind speciális konténer objektumok. A fentiek közül csak a tömb típus az Ada beépített típusa, a többit a programozónak kell megvalósítani, ha szüksége van rá. Ezek ugyanakkor sok esetben rugalmasabban használhatóak, mint a tömbök.

A következő fejezetben az általános `Lista`, mint absztrakt adattípus felépítésével mutatunk példát a konténer típusok készítésére.

13.1. A Lista típus

Egy lista egy olyan konténer, amelyben a tárolt objektumoknak létezik egy logikai sorrendje, s alapvetően a következő műveletek végrehajtására ad lehetőséget:[1]

- új objektum beszúrása a listába tetszőleges (logikai) helyre,
- egy objektum törlése a listából,
- a lista objektumainak felsorolása a logikai sorrendnek megfelelően.

Az absztrakt adattípusok megvalósítása Adában a csomagok és a sablon csomagok segítségével történik. A csomag specifikációs részében adjuk meg az (átlátszatlan) típusnevet, és az elérhető műveletek specifikációit, valamint a (rejtett) reprezentációt.

Milyen műveletekre van szükség? A teljesség igénye nélkül soroljuk fel a legfontosabbakat:

- a lista inicializálása – a típusinvariáns beállítása,

[1] A listában tárolt objektumok száma az elérhető memóriakapacitástól függ.

- a lista megszüntetése
- értékadás
- egyenlőségvizsgálat

A logikai sorrendben tárolt elemek elérését és kezelését teszik lehetővé a következő műveletek:

- ugrás a lista „elejére"
- ugrás a lista „következő" elemére
- ugrás a lista „megelőző" elemére
- ugrás a lista „végére"
- „aktuális elem" lekérdezése
- a lista „elején" állunk?
- a lista „végén" állunk?
- üres a lista?
- új listaelem beszúrása a lista „elejére" (hasonlóan lehetne a „végére" is)
- új listaelem beszúrása a lista aktuális eleme mögé (hasonlóan lehetne az aktuális elem elé, illetve az aktuális elem helyett is)
- az aktuális elem törlése a listából

A `Lista` típusban tárolható elemek száma a program futása közben gyakran változhat, így nem célravezető, hogy a reprezentációban tömb típussal, az objektum deklarálásakor fix méretben rögzítsük a lehetséges maximális elemszámot. Rugalmasabb kezelést tesz lehetővé a dinamikus objektumok használata. Ebben az esetben azonban, vagyis, ha egy típus reprezentációját dinamikusan allokált objektumok felhasználásával valósítjuk meg, fokozottan célszerű a típust „ellenőrzötten" kezelni, ügyelni arra, hogy egy objektum létrehozásakor a típusinvariáns biztosan igaz legyen, az objektum felszabadításakor az esetleg „hozzáláncolt" egyéb objektumok felszabadítása is biztosan megtörténjen, valamint, hogy értékadáskor ne a referenciákat adjuk csak át, hanem ha szükséges, hozzuk létre az eredeti objektum egy másolatát, s az erre vonatkozó referenciát tároljuk az értékadás bal oldalán álló entitásban. Az ilyen problémák kezelésére definiálták az `Ada.Finalization` csomagban a `Controlled` típust (7.8.).

Definiáljuk a `Lista` típust a `Controlled` típus leszármazottjaként! Ekkor a `Controlled` típus megfelelően felüldefiniált `Initialize` és `Finalize` műveletei az objektum létrehozásakor, illetve felszabadításakor automatikusan lefutnak. Az `l1:=l2` értékadáskor pedig az `Adjust` művelet felüldefiniált változatát fogja használni.

13.2. A Lista típus megvalósítása

A `Lista` típus – ahogy általában a konténerek típusok – független a benne tárolt objektumok típusától, ezekre csak az értékadás (és esetleg az egyenlőségvizsgálat) lehetősége szükséges, így megadhatjuk sablon csomagként, ahol a sablon paramétere a tárolt elemek típusa. A specifikáció tehát Adában:

13.1. Példa

```
with Ada.Finalization; use Ada.Finalization;
generic
type Elem is private;
package Listák is
  type Lista is new Controlled with private;
    -- a Controlled leszármazottja
  procedure Initialize(L: in out Lista);
  procedure Finalize(L: in out Lista);
  procedure Adjust(L: in out Lista);
  function "=" (L1, L2: Lista) return Boolean;
    -- egyenlô-e két lista?
  procedure Elejére(L: in out Lista);
    -- a lista elsô eleme lesz az aktuális,
    -- ha nem üres a lista
  procedure Végére(L: in out Lista);
    -- a lista utolsó eleme lesz az aktuális,
    -- ha nem üres a lista
  function Akt_Elem(L: in Lista) return Elem;
    -- visszaadja a lista "aktuális" elemét, ha
    -- van, egyébként kiváltja az Üres, vagy a
    -- Vége kivételt
  function Elején(L: Lista) return Boolean;
    -- igaz, ha a lista elsô eleme az aktuális
  function Végén(L: Lista) return Boolean;
    -- igaz, ha a lista utolsó eleme az aktuális
  function Üres_E(L: Lista) return Boolean;
    -- igaz, ha a lista üres
  procedure Következôre(L: in out Lista);
    -- az aktuális elem
```

```
    -- a logikailag következô elem lesz
  procedure Elôzôre(L: in out Lista);
    -- az aktuális elem
    -- a logikailag megelôzô elem lesz
  procedure Beszúr_Elejére(L: in out Lista; T: in Elem);
    -- az adott elemet beszúrja
    -- a logikailag elsô helyre ez lesz
    -- az új aktuális elem
  procedure Beszúr_Akt_Mögé(L: in out Lista; T: in Elem);
    -- az adott elemet logikailag
    -- beszúrja az aktuális mögé
    -- ez lesz az új aktuális elem
  procedure Töröl_Akt(L: in out Lista);
    -- törli az aktuális elemet a listából
    -- az új aktuális a logikailag Következô lesz, ha
    -- nem az utolsót töröltük, különben a
    -- megelôzô, ha van, ha az sincs,
    -- akkor üres lesz a lista
  Üres: exception;
    -- ha üres lista aktuális elemét akarjuk lekérdezni
  Vége: exception;
    -- ha a lista utolsó eleme utáni elemet akarnánk
private
  type Lista_Elem;
  type Lista_Elem_Mut is access all Lista_Elem;
  type Lista_Elem is
-- a lista elemeket kétirányú láncolással valósítjuk meg
  record
   Érték: Elem;
   Elôzô: Lista_Elem_Mut;
   Következô: Lista_Elem_Mut;
  end record;
  type Lista is new Controlled with
  record
   Elsô: aliased Lista_Elem_Mut:=null;
   Utolsó: aliased Lista_Elem_Mut:=null;
   Aktuális: aliased Lista_Elem_Mut:=null;
  end record;
end Listák;
```

A csomag törzsében adjuk meg a műveletek implementációját. Az elemek felszabadítására példányosítjuk az `Unchecked_Deallocation` eljárássablont a `Lista_Elem`, és a `Lista_Elem_Mut` típusok segítségével.

13.2. Példa

```
with Unchecked_Deallocation;
package body Listák is
  procedure Elem_Felszab is
  new Unchecked_Deallocation(Lista_Elem, Lista_Elem_Mut);

  procedure Initialize(L: in out Lista) is
  begin
   L.Elsô:=null;
   L.Utolsó:=null;
   L.Aktuális:=null;
  end;

  procedure Felszabadit(L :in out Lista) is
 -- az Elem_Felszab segítségével felszabadítja
 -- a teljes listát
   Akt:Lista_Elem_Mut:=L.Elsô;
   Tmp:Lista_Elem_Mut;
  begin
   while Akt /= null loop
     Tmp:=Akt;
     Akt:=Akt.Következô;
     Elem_Felszab(Tmp);
   end loop;
  end Felszabadit;

  procedure Finalize(L :in out Lista) is
  begin
   if L.Elsô /= null then Felszabadit(L);
     L.Elsô:= null;
   end if;
  end;
```

Az `Adjust` művelet implementációjánál két lehetőség között választhatunk:

- az ún „felszíni másolás"[2] létrehoz egy új listát, s másolja a `Lista_Elem` típusú objektumokat. Ekkor az `Érték Elem` típusú komponensből létrejön egy új példány. Ha ez nem egy változóra vonatkozó referencia, akkor így két független listát kapunk, ha azonban az `Elem` típusa referencia, akkor a két lista a „mélyben" ugyanarra az objektumra fog hivatkozni.

[2](shallow copy)

- a „teljes másolás" [3] esetén az `Elem` típusú objektumból teljes mélységében létrehozunk egy másolatot.

Az, hogy ténylegesen melyikre van szükség, mindig az aktuális alkalmazástól is függ. Most a `Lista` típushoz a felszíni másolást valósítottuk meg.

13.3. Példa

```
procedure Adjust(L: in out Lista) is
  Akt: Lista_Elem_Mut:=L.Elsô;
 -- az eredeti lista elejérôl indul
  Utols: Lista_Elem_Mut:= null;
 -- az utoljára létrehozott elem mutatója
  Eloz: Lista_Elem_Mut:=null;
 -- az elôtte létrehozott elem mutatója
  Els: Lista_Elem_Mut:=null;
 -- az Elsô elem mutatója
  Ujakt: Lista_Elem_Mut:=null;
 begin
  while Akt/=null loop
    Utols:=new Lista_Elem'(Akt.Érték,Eloz,null);
    if Els=null then Els:=Utols; end if;
    if Eloz /=null then Eloz.Következô:=Utols; end if;
    if Akt=L.Aktuális then Ujakt:=Utols; end if;
    Eloz:=Utols;
    Akt:=Akt.Következô;
  end loop;
  L.Elsô:=Els;
  L.Utolsó:=Utols;
  L.Aktuális:=Ujakt;
 end;
```

Mikor mondunk két listát egyenlőnek? Erre a kérdésre valójában háromféle „mélységű" válasz lehetséges:

A két lista pontosan ugyanazokat a listaelemeket tartalmazza, s az értékek logikai sorrendje is megegyezik.

A két lista pontosan ugyanazokat a listaértékeket tartalmazza, s az értékek logikai sorrendje is megegyezik. (Ez felel meg a „felszíni másolás" egyenlőségének.)

[3] (deep copy)

A két lista pontosan ugyanazt a szerkezetet írja le, s az értékek logikai sorrendje a „mélyben" is megegyezik. (Ez felel meg a „teljes másolás" egyenlőségének.)

Most a „felszíni" egyenlőség megvalósítását választottuk, összhangban az Adjust művelettel:

13.4. Példa

```
function "=" (L1, L2: Lista) return Boolean is
 L1_Mut: Lista_Elem_Mut:=L1.Elsô;
 L2_Mut: Lista_Elem_Mut:=L2.Elsô;
begin
 while L1_Mut/=null and L2_Mut /=null loop
   if L1_Mut.Érték/=L2_Mut.Érték then
    return False;
   end if;
   L1_Mut:=L1_Mut.Következô;L2_Mut:=L2_Mut.Következô;
 end loop;
 return L1_Mut=L2_Mut;
end;
```

A következő műveletek a lista bejárását, és a logikailag aktuális elem lekérdezését teszik lehetővé. Az aktuális elem lekérdezésekor, az Üres kivételt váltja ki, ha a lista még üres, s a Vége kivételt, ha az utolsó elem utáni elemet kellene visszaadni (pl. a Következôre művelet a lista végére való alkalmazása esetén).

13.5. Példa

```
procedure Elejére(L: in out Lista) is
begin
  L.Aktuális:=L.Elsô;
end;

procedure Végére(L: in out Lista) is
begin
  L.Aktuális:=L.Utolsó;
end;

function Akt_Elem(L:in Lista) return Elem is
begin
  if L.Aktuális/=null then
    return L.Aktuális.Érték;
  elsif L.Elsô=null then
    raise Üres;
```

```
    else raise Vége;
    end if;
  end;

  function Elején(L:Lista) return Boolean is
  begin
    return L.Aktuális=L.Elsô;
  end;

  function Végén(L:Lista) return Boolean is
  begin
    return L.Aktuális=L.Utolsó;
  end;

  function Üres_E(L:Lista) return Boolean is
  begin
    return L.Elsô=null;
  end;

  procedure Következôre(L:in out Lista) is
  begin
    L.Aktuális:= L.Aktuális.Következô;
  end;

  procedure Elôzôre(L:in out Lista) is
  begin
    L.Aktuális:=L.Aktuális.Elôzô;
  end;
```

A következő műveletcsoport csak ízelítő a lehetőségekből. Egy új listaelemet be lehetne szúrni a lista „elejére", a „végére", a lista aktuális eleme mögé, elé, illetve helyett, a megvalósítás hasonló lenne. Ugyanígy nem csak az aktuális elemet lehetne törölni, hanem az elsőt, az utolsót, az *i*.-et, az `x` értékűt stb. is. Törlésnél lehetne olyan megvalósítás mellett is dönteni, ami kivételt jelez, ha a törlendő elem nem szerepel a listában.

Ezek a műveletek változtatják a lista aktuális elemének pozícióját is. A választott megvalósítás beszúrásnál mindig az új elemet teszi aktuálissá, törlésnél az új aktuális a logikailag következő lesz, ha nem az utolsót töröltük, különben a megelőző, ha van, s ha az sincs, akkor üres lesz a lista (nem jelez hibát).

13.6. Példa

```
procedure Beszúr_Elejére(L: in out Lista; T: in Elem) is
   Uj: Lista_Elem_Mut;
```

```
begin
  if L.Elsô=null then
    Uj:=new Lista_Elem'(T,null,null);
    L.Elsô:=Uj;L.Utolsó:=Uj;
  else
    Uj:=new Lista_Elem'(T,null,L.Elsô);
    L.Elsô.Elôzô:=Uj;
    L.Elsô:=Uj;
  end if;
  L.Aktuális:=Uj;
end;

procedure Beszúr_Akt_Mögé(L: in out Lista; T: in Elem) is
  --és ez az új Aktuális
  Uj: Lista_Elem_Mut;
begin
  if L.Elsô=null then --Üres lista
    Uj:=new Lista_Elem'(T,null,null);
    L.Elsô:=Uj;
    L.Utolsó:=Uj;
  else
    Uj:=new Lista_Elem'(T,L.Aktuális,L.Aktuális.Következô);
    if L.Aktuális=L.Utolsó then L.Utolsó:=Uj; end if;
    L.Aktuális.Következô:=Uj;
    if  Uj.Következô/=null then --nem Utolsó volt
      Uj.Következô.Elôzô:=Uj;
    end if;
  end if;
  L.Aktuális:=Uj;
end;

procedure Töröl_Akt(L: in out Lista) is
  Akt:Lista_Elem_Mut:=L.Aktuális; --a törlendô elem
begin
  if Akt/=null then
    if Akt.Elôzô/=null then
      Akt.Elôzô.Következô:=Akt.Következô;
    else
      L.Elsô:=Akt.Következô;
      if L.Elsô=null then L.Utolsó:=null; end if;
    end if;
    if Akt.Következô/=null then
      Akt.Következô.Elôzô:=Akt.Elôzô;
    else
```

```
      L.Utolsó:=Akt.Elôzô;
      if L.Utolsó=null then
        L.Elsô:=null;
      end if;
    end if;
    if Akt.Következô/= null then L.Aktuális:=Akt.Következô;
    elsif Akt.Elôzô/=null then L.Aktuális:=Akt.Elôzô;
    else L.Aktuális:=null;
    end if;
    Elem_Felszab(Akt);
  end if;
end;
```

A fenti csomag sablont, – mivel a `Controlled` típus leszármazottját definiálja – használat előtt önálló fordítási egységként kell példányosítani. Például `Character` típusra:

```
with Listák;
package Chlist is new Listák(Character);
```

Az így keletkezett csomag már tetszés szerint használható:

13.7. Példa

```
with Chlist, Text_Io; use Chlist,Text_Io;
procedure Listákpr is
  ....
  L1, L2: Lista;          -- karakterek listája
  Kar:Character;
begin
  Kar:='a';
-- ciklusban, vagy anélkül beszúrhatunk akárhány elemet:
  for I in 1..20 loop
    Beszúr_Akt_Mögé(L1,Kar);
    Kar:=Character'Succ(Kar);
  end loop;
  Beszúr_Elejére(L1,'b');
-- kiirathatjuk az aktuális elemet
  Put(Akt_Elem(L1));
-- tetszés szerint "navigálhatunk" a listában:
  Végére(L1);
  ....
  Elejére(L1);
  ....
  Következôre(L1);
```

```
  ....
-- törölhetjük az aktuális elemet:
  Töröl_Akt(L1);
  ....
-- értékül adhatjuk egyik listát a másiknak, ez most két
-- teljesen különbözô listát eredményez:
  L2:=L1;
  ....
-- végigmehetünk a listán a logikai sorrendnek megfelelôen:
  Elejére(L1);
  if not Üres_E(L1) then Put(Akt_Elem(L1)); end if;
  while not Végén(L1) loop
    Következôre(L1);
    Put(Akt_Elem(L1));
  end loop;
  ....
-- vagy fordított sorrendben:
  Végére(L1);
  if not Üres_E(L1) then Put(Akt_Elem(L1)); end if;
  while not Elején(L1) loop
    Elôzôre(L1);
    Put(Akt_Elem(L1));
  end loop;
  ....
end;
```

Az eddigiekből kitűnik, hogy az így definiált `Lista` típus elég általános, így számos más típus létrehozásánál is támaszkodhatunk rá – például definiálhatjuk a `Halmaz`, `Verem`, `Sor` típusokat is, mint a `Lista` típus leszármazottait.

13.3. Esemény-listák

Az előzőekben definiált `Lista` típus jól használható az objektum-orientált programozás támogatására is. Ekkor ui. gyakori igény, hogy az egyes osztályhierarchiáknak megfelelő heterogén szerkezeteket szeretnénk futási-dejű kiválasztással, de mégis valamilyen egységes elv szerint kezelni. Ezt legegyszerűbben úgy valósíthatjuk meg, ha a `Lista` típus `Elem` típusa példányosításkor az adott származtatási fa gyökerének osztályszintű típusára mutató típus. Ekkor, ha az osztályszintű műveleteket hívjuk a lista elemei által mutatott objektumokra, azok dinamikus összekapcsolással, futási időben választják ki az aktuális típusnak megfelelő műveletet.

Példaként tekintsük a 12. fejezetben megismert biztonságvédelmi rendszert. Az absztrakt Esemény típushoz a BV_Rendszer_Alap csomagban definiáltuk a

```
type Esemény_Mut is access all Esemény'Class;
```

mutató típust. Ezzel példányosítjuk a Listák csomagot:

```
with BV_Rendszer_Alap, Normál_Biztonságvédelmi_Rendszer,
     Listák;
use BV_Rendszer_Alap, Normál_Biztonságvédelmi_Rendszer;
package Esemény_Listák is new Listák(Esemény_Mut);
```

Az így kapott Esemény_Listák csomag Lista típusa segítségével azután tetszés szerint kezelhetjük a beérkező eseményeket:

13.8. Példa

```
with Esemény_Listák, BV_Rendszer_Alap,
     Normál_Biztonságvédelmi_Rendszer, Text_IO, Ada.Tags;
use Esemény_Listák, BV_Rendszer_Alap,
     Normál_Biztonságvédelmi_Rendszer, Text_IO, Ada.Tags;
procedure Esemény_Listák_próba is
...
 L1, L2: Lista; -- ezek Esemény'Class-beli objektumok
-- mutatóinak listái lesznek
 Ae: Alacsony_Szintû_Esemény;
 Ke: Közepes_Szintû_Esemény;
 Fe: Felsô_Szintû_Esemény;

 E1: aliased Esemény'Class:=Ae;
 E2: aliased Esemény'Class:=Ke;
 E3: aliased Esemény'Class:=Fe;

 Emut1,Emut2: Esemény_Mut;

-- például az Es_Kiir osztályszintü müvelettel dinamikusan
-- hívjuk az egyes leszármazottakban felüldefiniált
-- Naplóz müveletet
-- és kiiratjuk az aktuális paraméter címke információját:
procedure Es_Kiir(Ac: in Esemény'Class) is
  begin
 ....
   Naplóz(Ac);
```

```
   Put_Line(External_Tag(Ac'Tag));
end;

begin
-- mutatót állíthatunk a program változóira:
  Emut1:= E1'Unchecked_Access;
-- az Esemény jelölt típus
-- más, külsö csomagban lett definiálva, ezért
-- nem írhatunk itt csak E1'Access hivatkozást
-- (a felhasználó felelössége, hogy az E1 objektuma a program
-- végén felszabadul)

-- de dinamikusan is létrehozhatjuk a megfelelö objektumot
  Emut2:= new Közepes_Szintû_Esemény;
-- (természetesen az adott esetben ezeket ténylegesen
-- majd a megfelelô események bekövetkezése váltja ki)

-- a listába beszúrhatjuk a statikus változókat éppúgy, mint a
-- dinamikusan allokáltat:
  Beszúr_Elejére(L1, Emut1);
  Beszúr_Akt_Mögé(L1,Emut2);

-- de közvetlenül is átadhatjuk a változó címét:
  Beszúr_Elejére(L1,E2'Unchecked_Access);
```

A kialakult listát egységesen kezelhetjük, természetesen mindig ügyelni kell arra, hogy az osztályszintű műveletek a lista aktuális elemei által mutatott objektumokra vonatkoznak. Például:

13.9. Példa

```
 Elejére(L1);
 if not Üres_E(L1) then
   Emut1:= Akt_Elem(L1);
   Es_Kiir(Emut1.all);
 end if;

 while not Végén(L1) loop
  Következôre(L1);
  Es_Kiir(Akt_Elem(L1).all);
 end loop;
```

A használat során arra is figyelnünk kell, hogy az `Adjust` művelet fenti definiálása felszíni másolást eredményez, tehát például egy `L2:=L1;` értékadás után, bár két különböző listát kapunk, de a két lista elemei ugyanazokra az objektumokra hivatkoznak. A biztonságos kezelés érdekében egy

teljes rendszerben valószínűleg érdemes létrehozni a `Lista` típus egy olyan változatát is, amelyik az `Adjust` műveletben a teljes szerkezetet újra létrehozza, majd a példányosítandó elemtípustól függően választani az adott alkalmazásban szükségeset.

13.4. Feladatok

13.1. Írjon programot, mely információkat tárol egy `Épület` típusú objektumról! Az `Épület` különböző célú `Helyiség`-ek gyűjteménye. A `Helyiség` típust ld. 12.1. A műveletek tegyék lehetővé az információk tárolását, lekérdezését és a lehetséges (értelmes) módosításokat!

13.2. Valósítsa meg a `Bevásárlókosár` gyűjtemény típust átlátszatlan típussal! Az `Élelem` felsorolási típus írja le a `Bevásárlókosár` típusú változókban tárolható élelmiszerféleségeket. Mindegyikből `0` vagy több fér egy kosárba. Valósítsa meg a következő típusműveleteket:
`Kiürít` – visszaad egy üres kosarat,
`Beletesz` – betesz a kosárba egy élelmiszerből valahányat,
`Mennyi_Van` – megadja, hogy egy élelmiszerből hány van a kosárban,
`Van_Már_Mindenből` – van-e minden élelmiszerből már legalább `n` darab,
`+` – az egyik kosár teljes tartalmát átteszi a másikba,
`Kiír` – kiírja az adott kosárban lévő élelmiszerek nevét és darabszámát.

13.3. Valósítsa meg a tetszőleges elemeket tartalmazó `Halmaz` típust a szokásos halmazműveletekkel!

14. Fejezet

Adatfolyamok

Egy *adatfolyam* (`stream`) elemek olyan sorozata, amelyek nem feltétlenül azonos típusúak, és az elemekhez való hozzáférés szekvenciális.

Az adatfolyam típusokat az `Ada.Streams` csomag írja le, az adatfolyamok típusosztályának az absztrakt `Ada.Streams.Root_Stream_Type` típus a gyökér típusa.

Egy új adatfolyam típust a gyökér típus (vagy egy már definiált másik adatfolyam típus) kiterjesztésével hozhatunk létre, megadhatunk új olvasó (`Read`) és író (`Write`) műveleteket, és opcionálisan definiálhatunk egyéb primitív műveleteket is.

Az előre definiált `T'Read` és `T'Write` adatfolyamokra definiált attribútumok dinamikus összekapcsolással hívják a `Root_Stream_Type` típus megfelelő leszármazottjának `Read` és `Write` műveletét. (A felhasználó által definiált `T'Read` és `T'Write` attribútumok is így működnek.)

14.1. Az `Ada.Streams` csomag

Az `Ada.Streams` csomag írja le az adatfolyam típusosztály alapvető lehetőségeit.

14.1. Példa

```
package Ada.Streams is
pragma Pure(Streams); -- a müveletek nem változtatják
   -- az adatfolyam típusok
   -- belsö állapotát
 type Root_Stream_Type is abstract tagged limited private;
-- a konkrét adatfolyam típusok ezen típus leszármazottai
 type Stream_Element is mod implementation-defined;
 type Stream_Element_Offset is range implementation-defined;
 subtype Stream_Element_Count is Stream_Element_Offset
   range 0..Stream_Element_Offset'Last;
 type Stream_Element_Array is
```

```
 array(Stream_Element_Offset range <>) of Stream_Element;
-- meghatározatlan méretű tömb típus
 procedure Read(
  Stream : in out Root_Stream_Type;
  Item   : out Stream_Element_Array;
  Last   : out Stream_Element_Offset) is abstract;
-- Az olvasó művelet Item'Length hosszúságú
-- adatfolyamelemeket hoz át az adatfolyamból
-- az Item tömbbe.
-- Az utolsóként áthozott adatfolyamelem indexét a
-- Last adja meg. Ez csak akkor kisebb
-- Item'Last-nál, ha elértük az adatfolyam végét.
-- A konkrét leszármazottakban kell megadni az implementációt.
 procedure Write(
  Stream : in out Root_Stream_Type;
  Item   : in Stream_Element_Array) is abstract;
-- Az író művelet hozzáfűzi az Item paraméter aktuális
-- értékét a specifikált adatfolyamhoz.
-- A konkrét leszármazottakban
-- kell megadni az implementációját.
private ... -- nincs előírva a nyelvi szabványban
end Ada.Streams;
```

14.1.1. Adatfolyam-attribútumok

Az adatfolyam típusok használatához alapvető jelentőségűek az attribútumok. A `Write`, `Read`, `Output`, és `Input` attribútumok konvertálják az értékeket adatfolyammá, és rekonstruálják a tárolt elemeket. Egy `T` típus minden `S` altípusához definiálva vannak ezek az attribútumok.

`S'Write` – egy eljárást jelöl, ami kiírja az `Item` értékét az adatfolyamba:

```
    procedure S'Write(
        Stream:access Ada.Streams.Root_Stream_Type'Class;
        Item:in T)
```

`S'Read` – egy eljárást jelöl, ami beolvassa az `Item` értékét az adatfolyamból:

```
    procedure S'Read(
        Stream:access Ada.Streams.Root_Stream_Type'Class;
        Item:out T)
```

Az attribútumok reprezentációját elemi típusokra az implementáció definiálja, összetett típusokra a komponens típusok `Write` és `Read` attribútumait kanonikus sorrendben hívja, ez tömbökre sorfolytonos megvalósítást jelent, rekordokra pedig pozíció szerintit. Ha a `T` egy tömb típus, akkor a

határai nem szerepelnek az adatfolyamban, ha T egy diszkriminánssal rendelkező típus, akkor a diszkrimináns csak akkor szerepel, ha van előredefiniált kezdeti értéke. Ha T egy jelölt típus, a címke nem szerepel.

S'Output – egy eljárást jelöl, ami kiírja Item értékét az adatfolyamba, minden indexhatárral, diszkriminánssal, illetve jelölt típusok esetén a címke információval együtt:
```
procedure S'Output(
     Stream:access Ada.Streams.Root_Stream_Type'Class;
     Item:in T)
```

S'Input – egy függvényt jelöl, ami beolvas és visszaad egy értéket az adatfolyamból, használja az S'Output-tól kapott információkat a határokra, a diszkrimináns és címke értékre:
```
function S'Input(
     Stream:access Ada.Streams.Root_Stream_Type'Class)
     return T;
```

Minden T'Class osztályszintű típus S'Class altípusára is definiálva vannak ezek az attribútumok:

S'Class'Write – egy eljárást jelöl, ami dinamikus összekapcsolással hívja az Item paraméter aktuális címke értékének megfelelő Write attribútumot:
```
procedure S'Class'Write(
    Stream: access Ada.Streams.Root_Stream_Type'Class;
    Item: in T'Class)
```

S'Class'Read – egy eljárást jelöl, ami dinamikus összekapcsolással hívja az Item paraméter aktuális címke értékének megfelelő Read attribútumot:
```
procedure S'Class'Read(
    Stream: access Ada.Streams.Root_Stream_Type'Class;
    Item: out T'Class)
```

S'Class'Output – egy eljárást jelöl, ami először kiírja Item külső címkéjét, (String'Output(Tags.External_Tag(Item'Tag)) hívásával), majd dinamikus összekapcsolással hívja az Item paraméter aktuális címke értékének megfelelő Output attribútumot:
```
procedure S'Class'Output(
    Stream: access Ada.Streams.Root_Stream_Type'Class;
    Item: in T'Class)
```

S'Class'Input – egy függvényt jelöl, ami először elolvassa az adatfolyamból a külső címkét, meghatározza a megfelelő belsőt (ehhez meghívja a Tags.Internal_Tag(String'Input(Stream))-et), majd dinamikus

összekapcsolással hívja az így kapott aktuális címke értékének megfelelő `Input` attribútumot:

```
function S'Class'Input(
    Stream: access Ada.Streams.Root_Stream_Type'Class)
    return T'Class
```

A `Read` és az `Input` attribútumok az összetett típusokra vonatkozó minden előredefiniált implementációjánál, ha ezek diszkriminánst is tartalmaznak, ellenőrzik, hogy a kapott érték a megfelelő altípusba esik-e, és fellép a `Constraint_Error` kivétel, ha nem. Más skaláris típusokra nincs ellenőrzés.[1]

Bármely típusra definiálhatunk adatfolyam attribútumokat egy attribútum definiálási záradékkal. Ezeknek a műveleteknek minden nem korlátozott típusra létezik előre definiált implementációja, a korlátozott típusokra pedig definiálhatunk, ha szükséges.[2]

Az adatfolyamok input-outputját az `Ada.Streams.Streams_IO` csomag írja le. (10.6.)

14.2. Objektum-hierarchiák tárolása adatfolyamokban

Egy jelölt típus különböző leszármazottait az adatfolyamok segítségével tárolhatjuk. A következő példák a 12. fejezetben bevezetett biztonságvédelmi rendszer különböző szintű riasztásainak fájlban tárolását, illetve a tárolt információ visszanyerését mutatják be.

A példaprogramokhoz használni fogjuk az eseményeket leíró `BV_Rendszer_Alap`, és `Normál_Biztonságvédelmi_Rendszer` csomagokon kívül az `Ada.Streams` és az `Ada.Streams.Stream_Io` csomagokat.[3]

Tegyük fel, hogy érvényesek a következő deklarációk:

14.2. Példa

```
F: Ada.Streams.Stream_Io.File_Type;
S: Stream_Access;
Ae : Alacsony_Szintû_Esemény;
Ke : Közepes_Szintû_Esemény;
```

valamint legyen az `Es_Kiir` a `Naplóz` műveletet dinamikusan hívó osztályszintű művelet:

[1] Mutató típusokra nincs előírva, hogy az implementáció ellenőrizze, hogy megfelelő értéket nyer-e vissza. Ha ellenőrzi, akkor szintén `Constraint_Error` lép fel hiba esetén, ha nem, akkor hibás végrehajtásra számíthatunk.

[2] A felhasználó által definiált osztályszintű attribútumok nem öröklődnek.

[3] Ez nem sablon csomag, mint a szekvenciális, illetve a közvetlen hozzáférésű fájlok input_outputját kezelő `Ada.Sequential_Io` és `Ada.Direct_IO` sablon csomagok.

```
procedure Es_Kiir(Ac: in Esemény'Class) is
begin
   ....
   Naplóz(Ac);
   Put_Line(External_Tag(Ac'Tag));
end;
```

A `Create` eljárással létrehozzuk az adatfolyamot, s a `Stream` függvény adja vissza a megfelelő `Stream_Access` mutatót:

```
Create(F, Name=>"prob");
S:=Stream(F);
```

A különböző eseményeket kiírhatjuk az adatfolyamba például a `Write` attribútummal statikus módon:

```
Alacsony_Szintû_Esemény'Write(S,Ae);
Közepes_Szintû_Esemény'Write(S,Ke);
```

Majd az adatfolyam elejére állhatunk a `Reset(F,In_File);` paranccsal. A tárolt információ csak akkor nyerhető vissza biztonságosan, ha pontosan tudjuk, hogy az egyes tárolt események milyen típusúak, vagyis azt, hogy az első `Alacsony_Szintû_Esemény`, s a második `Közepes_Szintû_Esemény` típusú volt, akkor a megfelelő `Read` művelet hívásával visszakapjuk a tárolt értéket:

```
  Alacsony_Szintû_Esemény'Read(S,Ae1);
-- statikusan hívható a típus megfelelö mûvelete:
  Kezel(Ae1);
  Közepes_Szintû_Esemény'Read(S,Ke1);
-- statikusan hívható a típus megfelelö mûvelete:
  Kezel(Ke1);
```

A munka végén az adatfolyamot a `Close(F);` paranccsal lezárjuk.

A `Write` attribútum nem tárolja a jelölt típusinformációt, így ha például az előbb másodikként kiírt `Közepes_Szintû_Esemény` típusú objektumot is megpróbáljuk az `Ae: Alacsony_Szintû_Esemény;` típusú változóba beolvasni egy `Alacsony_Szintû_Esemény'Read(S,Ae);` utasítással, sajnos nem jelez hibát – hiszen a `Közepes_Szintû_Esemény` típus leszármazottja az `Alacsony_Szintû_Esemény` típusnak, – hanem beolvassa az adatfolyamból a tárolt információ elejét, annyit, amennyit az `Alacsony_Szintû_Esemény` típusú változó tartalmaz. A hiba csak akkor következik be, ha az adatfolyam tartalmaz további eseményeket is, s ezeket is szeretnénk beolvasni és

kezelni – ekkor ui. a `Közepes_Szintû_Esemény` típusú változóból megmaradt információ már "szemétként" értelmeződik.

Abban az esetben azonban, ha kiiratáskor a `Write` attribútum helyett a megfelelő osztályszintű `Output` attribútumot használjuk, például:

```
Alacsony_Szintû_Esemény'Class'Output(S,Ae);
Közepes_Szintû_Esemény'Class'Output(S,Ke);
```

tárolja a megfelelő típusinformációt. Ekkor visszaolvasáskor az osztályszintű `Input` attribútum segítségével már futásidejű kiválasztással hívhatjuk a szükséges kezelő műveletét, például a következő ciklusban:

```
Reset(F,In_File);
....
while not End_Of_File(F) loop
  Es_Kiir(Alacsony_Szintû_Esemény'Class'Input(S));
end loop;
```

Az előredefiniált attribútumokat szükség esetén át is definiálhatjuk.

Összefoglalásként elmondhatjuk, hogy a nyelv az adatfolyamok segítségével lehetőséget ad objektumhierarchiák tárolására, és a tárolt információ biztonságos visszanyerésére.

14.3. Feladatok

14.1. Az `Épület` típusú objektumok információkezelését (ld. 13.1) valósítsa meg adatfolyamok segítségével!

14.2. Az `Alkalmazott` típusú objektumok (ld. 12.2) halmazának (ld. 13.3) információkezelését valósítsa meg adatfolyamok segítségével!

14.3. Írja meg az 12. fejezetben definiált `Alakzat` típus-hierarchiába tartozó elemek halmazát (ld. 13.3) kezelő programot úgy, hogy az elemeket adatfolyamban tárolja!

15. Fejezet

Taszkok

Az Adában a párhuzamosság kifejezésének nyelvi eszköze a *taszk*.

Egy program végrehajtása az utasításainak szekvenciális végrehajtásával történik. Az utasításokat egyetlen „logikai processzor” hajtja végre. A taszkok olyan egységek, amelyek végrehajtása párhuzamosan történik. Úgy képzelhetjük el, hogy mindegyik taszkot egy saját „logikai processzor” hajt végre, amely a taszk törzsének utasításait a szekvenciális programoknál megszokott módon végzi el. Az egyes „logikai processzorok” egymástól függetlenek, leképezésük módja fizikai processzorokra a nyelv implementációjához tartozó kérdés.

Szekvenciális programokban egyidőben egyetlen utasítás áll végrehajtás alatt, míg a párhuzamos programokban a végrehajtás több szálon fut, egy időpillanatban a programnak több utasítása is lehet végrehajtás alatt. A különböző taszkok a szinkronizációs pontoktól eltekintve egymástól függetlenül működnek, tehát a taszkok aszinkron módon együttműködő programegységek.

A taszkok szinkronizálása a taszkok randevúival történik, amelyek egy hívást kiadó, illetve fogadó taszk között zajlanak le. A taszkok rendelkeznek ún. *belépési pontokkal*. A taszk belépési pontja hívható egy másik taszkból. A hívott taszkban a belépési ponthoz legalább egy `accept` utasítás tartozik, és a hívás hatására ezek közül valamelyik végrehajtódik. A belépési pontnak paraméterei is lehetnek, amelyek a taszkok közötti információátadásra szolgálnak.

Ebben a fejezetben a következő nyelvi elemekkel ismerkedünk meg: taszk egységek, belépési pontok, a taszkok közötti kapcsolatok leírásához szükséges utasítások (belépési pont hívása, `accept`, `delay`, `terminate`, `select` és `abort` utasítás). A fejezet tartalmazza a fenti fogalmakra vonatkozó leírásokat, szabályokat.

15.1. A taszk specifikációja és törzse

A taszkot egy taszk programegységgel definiáljuk, amely taszk specifikációból és taszk törzsből áll. A specifikáció és a törzs szerepe pontosan olyan, mint más programegységek esetében. A specifikáció a `task type` alapszavakkal kezdődhet. Így taszk típust definiálunk és ezt használhatjuk taszk objektumok deklarálására.

Példa taszk típusok és egyszerű taszkok specifikálására:

15.1. Példa

```
task type Erôforrás is
   entry Lefoglal;
   entry Felszabadít;
end Erôforrás;

task type Vezérlés is
   entry Olvas (C: out Character);
   entry Ír (C: in Character);
end Vezérlés;

task Termelô;

task Fogyasztó;

task Tároló is
   entry Betesz(C: in Character);
   entry Kivesz(C: out Character);
end Tároló;
```

A taszk típus értékei olyan taszkok, amelyeknek belépési pontjai, a specifikációs részben felsorolt belépési pontok, a taszkok törzsét pedig a `task body` definiálja.

A `type` alapszó nélküli specifikáció egyedi taszkot definiál, amit *névtelen taszk típusnak* is nevezünk. Az ilyen taszkdeklaráció ekvivalens azzal, hogy definiálunk egy taszk típust, és ehhez deklarálunk egy taszkot.

A taszk törzse definiálja a taszk végrehajtásának hatását. A taszk törzsének végrehajtása a taszk elindításának hatására történik meg. Példák taszk törzsekre:

15.2. Példa

```
task body Termelô is
C: Character;
begin
   loop
```

```
      Termel(C);
      Tároló.Betesz(C);
   end loop;
end Termelő;

task body Fogyasztó is
C: Character;
begin
   loop
      Tároló.Kivesz(C);
      Fogyaszt(C);
   end loop;
end Fogyasztó;
```

Egy taszknak lehet diszkrimináns része is. A diszkriminánsra ugyanazok a szabályok érvényesek, mint a tömb, illetve a rekord típus diszkriminánsaira. Ha egy taszknak van diszkriminánsa, akkor ez a taszk típus egy objektumának létrehozásakor értékelődik ki. A taszk objektum élettartama alatt a diszkrimináns értéke nem változik.

```
task type Azonosító (Diszkrimináns: Típus) is
   entry Deklarálás;
end Azonosító;
```

A diszkrimináns olyan paraméter, amely befolyásolja a taszk objektumok végrehajtását. Egy taszk típus objektumai ugyanazokat az utasításokat hajtják végre, azonban a diszkrimináns különböző értékeket vehet fel a különböző objektumoknál. Így az utasítások végrehajtása különbözhet a diszkrimináns kezdeti értéke miatt. Például, ha különböző tömbökbe író taszk objektumokat akarunk létrehozni, akkor a tömb nevét diszkriminánsként adhatjuk meg:

```
task type T(Tnév: access String) is
   entry Betesz(K: Character);
end T;
```

Diszkriminánsos taszk nem lehet egyedi taszk. Példányosításkor írhatjuk:

```
A: T(Tnév => Karakterek'Access);
B: T(Tnév => Számjegyek'Access);
```

ahol `Karakterek` és `Számjegyek` két `String`.

Egy taszk törzsét helyettesítheti a következő deklarálás:

```
task body Név is separate;
```

ami azt jelöli, hogy a törzs külön forrásfájlban szerepel. A taszk törzsét tartalmazó forrásfájl pedig a `separate(Taszkot_deklaráló_eljárásnév)` sorral kezdődik.

15.2. Taszk típusok, taszk objektumok

Minden taszk megfelel egy objektumnak, amelyet *taszk objektumnak* nevezünk. Mint minden objektumnak, a taszknak is van típusa. Az ugyanolyan típusú taszk objektumok ugyanazt az utasítássorozatot hajtják végre, önálló módon.

Ha a taszk objektum olyan objektum, vagy olyan objektumnak a komponense, amelyet

- egy objektumdeklarációban deklaráltunk, a taszk objektum értékét a deklaráció kiértékelése nyomán határozzuk meg;
- egy allokátor (`new`) kiértékelése nyomán hoztunk létre, a taszk objektum értékét az allokátor kiértékelésekor határozzuk meg.

A taszk típus korlátozott típus, azaz a taszk objektumokra nem alkalmazhatjuk az értékadás és az `=`, `/=` műveleteket, így műveleteinek halmaza lényegében a taszk attribútumokra korlátozódik. Tehát a taszk típusú objektumok értéke konstans a program futása során. Taszk típusú objektum nem lehet alprogram `out` típusú vagy sablon `in` típusú paramétere. Egy függvény visszaadhat egy taszk típusú objektumot címszerinti átadással. Egy taszk típusú objektumot ugyanúgy deklarálunk, mint bármilyen más típusú objektumot.

```
T1, T2: Taszktípus;
type Taszktömb is array(1 .. 10) of Taszktípus;
type Taszkcím is access Taszktípus;
Tc: Taszkcím := new Taszktípus;
```

Példák taszk változók deklarálására:

```
type P is access Vezérlés;
E: Erôforrás;
V: Vezérlés;
A: array (1 .. 10) of Vezérlés;
W: P := new Vezérlés;
```

Minden taszk objektumnak saját másolata van a taszk típusban deklarált változókról.

Egy taszk típushoz tartozó új taszk objektum – a többi objektummal párhuzamosan – végrehajtja a taszk törzsének utasításait.

Egy taszk objektum a taszk lokális változói mellett használhatja a taszk törzsén kívül deklarált globális változókat is.

15.3. Taszkok elindítása, végrehajtása

A taszk végrehajtását a taszk törzse definiálja. A végrehajtás kezdeti lépése a taszk elindítása (ez a deklarációs rész kiértékelését is magában foglalja).

A taszk elindítása az objektumot deklaráló programegység törzse deklarációs részének feldolgozása után, a törzs első utasításának végrehajtása előtt történik meg. Ha a deklaráció egy csomag specifikációs részében van, az elindítás a törzs deklarációs részének kiértékelése után és a csomag törzsének végrehajtása előtt történik meg. Az a taszk objektum, vagy objektum komponens, amit allokátorral hozunk létre, az allokátor kiértékelésekor kerül elindításra, az objektum nem taszk típusú komponenseinek inicializálása után kezdődik.

Ha egy taszk elindítása közben hiba történik, a taszk *komplett* állapotba kerül, az elindítás befejezése után a taszkot tartalmazó programegység törzsében az első utasítást közvetlenül megelőző helyen, illetve az allokátor kiértékelésének helyén fellép a `Tasking_Error`.

Egy *komplett* állapotba került taszkot már semmilyen módon nem lehet újra elindítani.

15.4. Taszkok terminálása

A taszk objektumok függőségi relációban állnak más programegységekkel. Minden taszk függ legalább egy szülőtől. A szülő lehet taszk, végrehajtás alatt álló blokk, alprogram vagy könyvtári csomag. A függés közvetlen az alábbi két esetben.

- A taszk szülője az a taszk(objektum), blokk, alprogram vagy könyvtári egység, amely a taszk objektum deklarációját tartalmazza.
- Ha egy taszk allokátor kiértékelésével jött létre, akkor a mutató típus deklarációját tartalmazó taszk, blokk, alprogram vagy könyvtári egység a taszk szülője.

Ha egy taszk függ egy blokktól vagy alprogramtól, akkor a blokkot végrehajtó, illetve az alprogramot hívó taszktól, blokktól, alprogramtól vagy könyvtári egységtől is függ (közvetve). Ha egy `T1` taszk függ a `T2` taszktól, akkor a `T1` leszármazottja a `T2`-nek. Ha a `T1` taszk függ a `T2` taszktól, a `T2` pedig a `T3` taszktól, akkor a `T1` is függ a `T3`-tól.

A taszk *komplettté* válik, ha véget ér a taszk törzsében levő utasítások végrehajtása, vagy ha a végrehajtás közben hiba lép fel, és nincs a taszkban hibakezelő programrész, vagy ha van, és annak a végrehajtása befejeződik.

Egy taszk *terminál*, ha:

- komplett, és az összes tőle függő taszk már terminált.
- végrehajtása egy `terminate` utasításhoz ért és a taszk olyan programegységtől függ, amely már komplett (a végrehajtása befejeződött, tehát pl. nem könyvtári csomag) és a szülőtől függő összes leszármazott terminált már, komplett vagy szintén egy `terminate` utasításnál várakozik.
- `abort` következtében abnormális állapotba jutott, kompletté vált és törlődött a várakozási sorokból.

15.3. Példa

```
declare
type Globális is access Erôforrás;
   A, B: Erôforrás;
   G: Globális;
begin  -- elindul A és B
declare
type Lokális is access Erôforrás;
   X: Globális := new Erôforrás;
   L: Lokális := new Erôforrás;
   C: Erôforrás;
begin    -- elindul a C
   G := X;   -- G és X ugyanazt az objektumot jelölik
end; -- várakozás C és L.all terminálására
end;  -- várakozás A, B és G.all terminálására
```

A függőségi szabályok következtében könyvtári egységekbe ágyazott taszkok terminálása nem feltétlenül előzi meg a főprogramét (ki nem ürített I/O pufferek). A legkülső szinten deklarált taszkokhoz rendelt tárterület nem szabadítódik fel teljesen terminálásuk után, mert lehet rájuk aktív hivatkozás (pl.: `T'Callable`). Ennek következtében a rendelkezésre álló szabad memória elfogyhat (`Unchecked_Deallocation` nem használható taszkokra).

Nézzük meg figyelmesen az alábbi programrészletet!

15.4. Példa

```
with text_io; use text_io;
procedure Példa is
   task type T;
   type P is access T;
   task body T is begin null; end T;
   function F return P is
```

```
      X: P := new T;
   begin
      return X;
   end F;

begin
if F'Terminated then Put_Line("Terminált!");
   else Put_Line("Nem terminált!");
end if;
end Példa;
```

Mit kell kiírnia a programnak?

15.5. A taszkok attribútumai

A taszkokhoz is tartoznak attribútumok:

`TaskName'Callable` – logikai érték, ami pontosan akkor hamis, ha a taszk már komplетté vált, terminált vagy abortált (vagyis már nem lesz többé hívható).

`TaskName'Terminated` – logikai érték, ami pontosan akkor igaz, ha a taszk már terminált.

`EntryName'Count` – a megnevezett `entry`-hez tartozó `accept`-re várakozó hívások száma.

15.6. Belépési pont, belépési pont hívás, accept utasítás

A taszkoknak lehetnek belépési pontjaik, melyek az eljáráshívásokhoz hasonló entitások. Egy taszk kapcsolatban lehet egy másik taszkkal, azáltal, hogy meghívja azt (hasonlóan az eljáráshíváshoz).

A taszkok szinkronizálására és adatcseréjére elsősorban a belépési pont hívás és az `accept` utasítás szolgálnak. Az `entry` deklaráció hasonló az alprogram deklarációhoz. A belépési pont deklarálás tartalmazhat paramétereket, a belépési pont formális és aktuális paramétereire ugyanaz érvényes, mint az alprogramok paramétereire, azzal a különbséggel, hogy a belépési pontnak nem lehet `access` típusú paramétere. A belépési pont specifikációja szabja meg a neki megfelelő belépési ponton történő kommunikáció irányát és az átadott üzenetek típusát.

Például, ha egy `Taszk_Példa` taszknak van egy `Belépés` nevű belépési pontja, akkor ezt a következőképpen hívhatjuk meg:
`Taszk_Példa.Belépés(Kód);` ahol `Kód` a belépési pont paramétere.

Az `accept` utasítás definiálja azokat a műveleteket, amelyeket akkor kell elvégezni, ha a belépési pont hívása megtörtént. Egy `entry`-hez több `accept` utasítás is tartozhat.

Az `accept` segítségével egy taszk kommunikálhat közvetlen módon egy másik taszkkal. Ha egy taszk meghívja egy másik taszk belépési pontját és a másik taszk válaszol a hívásra, akkor randevú jön létre a két taszk között. Amikor két taszk kommunikálni akar egy randevú alkalmával, akkor az első taszk, amely a találkozási ponthoz ér, vár addig, amíg a másik is a találkozási ponthoz ér.

Amikor egy taszk hívást intéz egy másik taszkhoz, és a hívott taszk végrehajtása még nem jutott el a megfelelő `accept` utasításhoz, akkor a hívó felfüggesztődik.

Ha egy taszk végrehajtása egy `accept` utasításhoz ért, és nincs a megfelelő belépési pontra hívás, akkor a taszk felfüggesztődik.

Ha egy belépési pont a hívás pillanatában a hívott taszk a belépési ponthoz tartozó `accept` utasítás végrehajtásánál tart, akkor végrehajtódik az `accept` utasítás törzse és a hívó taszk felfüggesztődik. Ezáltal valósul meg a két taszk között a *randevú*. Az `accept` törzs végrehajtása után mindkét taszk párhuzamosan fut tovább, folytatják egymástól független módon az utasítások végrehajtását.

Egy taszk saját belépési pontjának hívása holtpontot eredményez!

Ha több taszk hívja ugyanazt a belépési pontot, miközben a végrehajtás nem érte el az `accept` utasítást, a hívások várakozási sorba kerülnek. Minden belépési ponthoz külön várakozási sor tartozik. Egy adott belépési ponton kommunikálni kívánó taszkok érkezési sorrendben várakoznak (FIFO). A megfelelő `accept` utasítás minden egyes végrehajtása kivesz egy várakozót a sorból. A várakozási sorból két módon törlődnek elemek: a legrégebben várakozó taszk hívásának fogadásával, ha a hívott taszk a belépési ponthoz tartozó `accept` utasításhoz ér (sikeres randevú), illetve a várakozásra szabott idő leteltével, vagy a várakozó taszk abortálása következtében (elmaradt randevú). Adott pillanatban egy belépési pontnál várakozó taszkok száma meghatározható az `E'Count` várakozási sor attribútum segítségével.

Példa az `accept` utasítás használatára:

```
task body Tároló is
T: Character;
begin
   loop
      accept Betesz(C: in Character) do
         T := C;
      end Betesz;
```

```
      accept Kivesz(C: out Character) do
         C := T;
      end Kivesz;
   end loop;
end Tároló;
```

Az accept utasítás része a taszk törzsében található utasítások sorozatának. Hívás esetén az az accept utasítás hajtódik végre, amelyhez a hívott fél hamarabb érkezett.

Egy accept utasítás paraméterei megegyeznek a belépési pontban található paraméterekkel.

A randevú esetén a két taszk a paraméterek segítségével kommunikálhat, értékeket adhat át egymásnak. Ha az accept utasítás paramétere in típusú akkor a paraméter értéke bemásolódik a formális paraméterbe. Ha in out típusú, akkor az eredmény visszamásolódik a hívó fél megfelelő aktuális paraméterébe.

Amíg az accept törzsének végrehajtása be nem fejeződik, addig a hívó fél várakozik.

Minden olyan tevékenységet, amelyet egy taszk végrehajthat a randevún kívül is, az accept törzsén kívül kell elhelyezni, azért, hogy a hívó fél ne várjon sokat tevékenységének folytatására.

Egy accept utasítást felhasználhatunk arra is, hogy a hívó tevékenységét szabályozzuk. Egy paraméter nélküli accept utasítással, melynek törzsében nem szerepel utasítás, elérhetjük azt, hogy a hívó taszk csak a hívott fél randevú fogadása után folytassa tevékenységét.

Egy T taszk accept utasítása csak a T taszk törzsében szerepelhet, nem szerepelhet alprogramban, csomagban, vagy olyan taszkban, amely be van építve a T-be.

Lényeges megjegyezni az eljáráshívás és a randevú közötti különbségeket. *Eljáráshívás* esetén a hívott eljárás törzse a hívó eljárás kódjának részévé válik. Az eljárás lokális változói megfelelő lokális változókként jönnek létre a hívóban. Egyszerre több folyamat is végrehajthatja ugyanazt az eljárást, ugyanazon eljárásnak egyidejűleg több példánya is lehet aktív (azaz az eljárás *reentráns programegység*).

Az eljárás számára globális változók (pl. az eljárást tartalmazó programegység deklarációs részében definiált objektumok) ilyenkor egyidejűleg több folyamat által is elérhetőek, kölcsönös kizárás nincs biztosítva!

A taszk nem reentráns programegység. A taszk törzs deklarációs részében definiált objektumokra a kölcsönös kizárás automatikusan biztosított. Randevú esetén két aktív folyamat kapcsolódik össze. Az interakció szinkron szimmetrikus módon megy végbe, azaz akkor, ha mindkét fél készen áll rá - a hívó taszk meg kell várja, amíg a hívott kész az ő fogadására.

15.7. Belépésipont-családok

A belépési pont deklaráció deklarálhatja belépésipont-családját is. Egy belépésipont-család tulajdonképpen belépési pontokból álló tömb. Segítségével olyan taszkokat írhatunk, amelyeknek ugyanaz a törzse, azonban a neve különbözik. Ezeket egyetlen belépési pontként írhatjuk meg és, hasonlóan a tömb elemeihez, a belépési pontokat az index különbözteti meg egymástól. A családot diszkrét intervallummal jelöljük ki. A család egy belépési pontjára indexeléssel hivatkozhatunk. A törzsön kívül a belépési pontok nevére szelekciós formában kell hivatkozni, ahol az előtag a taszk objektum neve. A családok alkalmazásával növelhetjük a belépési pontok számát és vele együtt a várakozási sorok számát. Egy várakozási soron belül csak érkezési sorrendben fogadhatóak a hívások, több várakozási sor esetén azonban ettől eltérő kiszolgálási sorrend is lehetséges. Egy családot a következőképpen deklarálhatunk:

```
task type Típus is
   entry Család(Diszkrét_típus) (Paraméterek);
end T;
```

Majd a `Család` meghívása a következőképpen történik:

```
T.Család(Kifejezés) (Paraméterek);
   -- a Kifejezés Diszkrét_típus-ú kell legyen
```

15.8. A `select` utasítás

Az eddig leírt randevú-mechanizmus lehetővé teszi két taszk kommunikálását, azonban ez a módszer egy kissé nehézkes, mert egy taszk addig kell várjon, amíg egy másik taszk nem fogadja hívását és fordítva, ha egy taszk fogadhat egy hívást, akkor addig vár amíg egy másik taszk meg is teszi a hívást.

A randevú mechanizmusát könnyíti a `select` utasítás kombinálva az `accept`, `delay`, `terminate` és `abort` utasításokkal. A kombinálás lehetővé teszi, hogy egy taszk egy adott ponton többféle hívást várjon, és elfogadja azt, amelyik hamarabb érkezik. Ugyanakkor különböző feltételek segítségével lehetővé teszi a hívások szabályozását. A `select` utasítás biztosíthatja azt is, hogy egy hívott taszk automatikusan termináljon, valamint lehetővé teszi, hogy egy taszk folytassa tevékenységét, ha bizonyos idő elteltével nem valósul meg a randevú.

A `select` utasítás különböző jelentéssel bír a hívó, illetve a fogadó taszkban.

Ezáltal a `select`, együtt az `accept`, `delay`, `terminate` utasításokkal különféle helyzetek megoldására alkalmazható, mint például: a taszkok szelektív fogadása; a szelektív várakoztatás, a feltételes `entry` hívás és az időhöz kötött `entry` hívás.

15.8.1. A szelektív várakoztatás

A taszkok hívásainak szelektív elfogadása

Tegyük fel, hogy számjegyeket és betűket akarunk kiírni a képernyőre. Ehhez szükségünk van arra, hogy a számjegyet, illetve a betűket kiíró taszkoknak szelektív módon engedélyezzük a kiírást a képernyőre.

15.5. Példa

```
task Engedély is
   entry Számjegy(I: in Integer);
   entry Karakter(K: in Character);
end Engedély;
```

A szelektív elfogadással ez úgy oldható meg, hogy a `select` utasításba írt `accept` ágak közé az `or`-t tesszük.

```
task body Engedély is
begin
   loop
      select
         accept Számjegy(I: in Integer) do
         put(Integer'Image(I));
         end Szám;
      or
         accept Karakter(K: in Character) do
         put(K);
         end Karakterlánc;
      end select;
      new_line;
   end loop;
end Engedély;
```

Egy taszk egyszerre kettőnél több belépési pontjára vonatkozó hívást is tud figyelni, ez azt jelenti, hogy a taszk `select` utasításában több `accept` ág lehet, és a különböző alternatívákat az `or` választja el egymástól.

A fenti példában a taszk, amikor a `select` utasításhoz ér megvárja, hogy valamelyik belépési pontja meghívásra kerüljön.

Ha egy taszk meghívta valamelyik belépési pontját, akkor a neki megfelelő `accept` alternatívát hajtja végre. Egy hívás akkor hajtható végre, ha a taszk elérte a `select` utasítást.

A taszkok hívásainak szelektív elfogadását megvalósító `select` utasítást végtelen ciklusba helyezzük, ezzel az `entry`-kben definiált műveleteket a taszk végre tudja hajtani kérés esetén. A taszk minden egyes lépés után végrehajt egyetlen alternatívát, azaz egyetlen `accept` ágat. Ha a `select` utasításban szereplő belépési pontok közül egynél többhöz tartozik nem üres várakozási sor, akkor a végrehajtásra kerülő `accept` utasítás kiválasztása nem-determinisztikus.

Feltételes hívásfogadás

A feltételes hívásfogadás esetén a különböző `select` alternatívákat fogadási feltételekkel egészítjük ki. A `when feltétel =>` szerkezettel szabályozhatjuk azt, hogy milyen esetben fogadja a taszk egy `entry`-vel deklarált belépési pont hívását.

Egy ágat *nyitottnak* nevezünk, ha nincs benne feltétel (`when`), vagy ha a feltétel értéke igaz, különben *zártnak* nevezzük. Zárt ág nem választható ki.

Feltételes hívásfogadás esetén egy taszk csak azokat a hívásokat tudja fogadni, amelyeknek egy nyitott ág felel meg a `select` utasítás végrehajtásának kezdetén, mivel ekkor értékelődnek ki a belépési pontok feltételei egyetlen egyszer. Az őrfeltételek ismételt kiértékelésére akkor kerül sor, ha a `select` utasítás egy ciklus magjában helyezkedik el.

Ha minden ág zárt, akkor kivétel lép fel.

A `terminate` alternatíva

A taszkok többségénél a `select` utasítást egy végtelen ciklus tartalmazza, ami azt jelenti, hogy a taszk törzse soha nem ér véget, a taszk nem adja át a vezérlést másik programegységnek. Ezáltal az eljárás, amelyben a taszk objektumot deklaráltuk, soha nem ér véget. A `select` utasítás `terminate` alternatívája biztosítja, hogy a taszk törzsének végrehajtása lezárul. A `terminate` alternatívát használhatjuk feltétellel
(`when feltétel => terminate;`) vagy pedig feltétel nélkül.

Miután egy taszk minden taszk hívását fogadta a taszk specifikációjában megadott belépési pontok szerint, és minden várakozási sora üres, akkor végrehajtja a `terminate` alternatívát.

```
select
   accept A1(X: in Jel; Y: out Üzenet) do
   -- olvassuk az X-et és az Y üzenetet küldjük
   end A1;
```

```
or
   accept A2(Z: in Üzenet; V: out Jel) do
   -- fordított művelet
   end A2;
or
   terminate;  -- a taszk terminál ha nem érkezik sem jel,
               -- sem üzenet
end select;
```

A `delay` és az `abort` utasítás

A `delay t;` utasítás felfüggeszti a taszkot *t* másodpercre. Az Ada a főprogramot magát is egy taszknak fogja fel, ezért a `delay` utasítás használható taszkon kívül is, ekkor a főprogramot függeszti fel az adott időre. Az idő megadására az Ada bevezeti a predefinit valós `Duration` típust. Az idő kezeléséhez az Ada könyvtár predefinit egységként tartalmazza a `Calendar` csomagot.

Egy taszk `select` utasításában használt késleltetés lehetővé teszi, hogy a taszk végrehajtson valamilyen műveletet, hogyha `t` időn belül nem választott ki semmilyen más `select`-alternatívát. Az időt a `select` utasítás megkezdésétől számítja. A `delay` alternatívát is valamilyen feltételhez köthetjük.

A `select` utasításban a `delay`-t akkor használjuk leginkább, amikor a taszk valamit periodikusan kell végrehajtson, de ugyanakkor taszkok hívását is kell fogadja.

Az `abort T1,T2;` utasítás *abnormális* állapotba hozza a megnevezett taszkokat. Ebben az állapotban a taszk már nem vehet részt további randevúban, így esetleg a lekötött erőforrások sem szabadulnak fel. Az abortált taszk az első szinkronizációs pont (belépési pont hívás, másik taszk indításának kezdete, saját aktivizálásának vége, `select`, `abort`, `delay` utasítás, kivételkezelő vagy `accept` utasítás kezdete és vége) elérése előtt komplettté válik. Egy taszk önmagát is abortálhatja (öngyilkos taszk).

Az `else` alternatíva

A `select` utasítás egy másik alternatívája az else ág. Az else ágon levő utasításokat akkor hajtja végre, ha a taszk semmilyen belépési pont hívást nem tud kiszolgálni azonnal (azaz a hívás megtörtént, de a neki megfelelő `accept` ág nem nyitott a fogadására vagy létezik nyitott ág, de nincs neki megfelelő hívás).

Az else ág olyan esetekben hasznos amikor egy taszk bizonyos utasítások ismételt végrehajtása mellett időnként belépési pont hívásokra is válaszol. Az else ágat addig hajtja végre, amíg nincs `entry` hívás. Belépési pont hívás esetén a megfelelő `accept`-et hajtja végre.

Összefoglalásként elmondhatjuk, hogy egy hívott taszkban a `select` utasítás lehetővé teszi valamely feltételek egyikének teljesüléséig a várakozást, majd egy feltétel teljesülésekor a hozzá tartozó utasítások végrehajtását.

A szelektív várakoztatásnak legalább egy olyan ága kell legyen, ami `accept` utasítást tartalmaz, ezen kívül lehet `delay` utasítást és `terminate` utasítást tartalmazó ága (a két ág közül egy időben csak egy lehet jelen).

A `select` utasítás végrehajtásakor a feltételek kiértékelésre kerülnek, és meghatározásra kerül, mely ágak nyitottak. Ezután végrehajtódik egy nyitott ág, vagy ha ilyen nincs, az `else` - ág. Ezzel az utasítás végrehajtása befejeződik. Ha több nyitott ág is van, a választás a következő szabályok szerint történik:

- ha van nyitott `accept` ág, amelynél randevú jön létre, az hajtódik végre;
- ha nincs nyitott `accept` ág, akkor egy nyitott `delay` ág választódhat ki;
- az `else` - ág csak akkor választódhat ki, ha mindegyik ág zárt;
- a `terminate` ág nem választódhat ki addig, amíg van a taszknak nem üres `entry`-sora.

Példa a szelektív várakoztatásra:

15.6. Példa
```
task body Erôforrás is
   Foglalt: Boolean := False;
begin
   loop
      select
         when not Foglalt => accept Lefoglal do
            Foglalt := True;
         end Lefoglal;
      or
         accept Felszabadít do
            Foglalt := False;
         end Felszabadít;
      or
         terminate;
      end select;
   end loop;
end Erôforrás;
```

A különböző belépési pontokhoz tartozó accept utasítások egymásba ágyazhatóak, illetve egy accept utasítás törzse tartalmazhat belépési pont hívást (*egymásba ágyazott randevú*).

Egy adott belépési ponthoz tartozó randevú befejeződéséig ugyanezen belépési ponthoz tartozó újabb randevú azonban nem jöhet létre (azonos belépési ponthoz tartozó accept utasítások nem ágyazhatóak egymásba). A belső randevúra esetleg nagyon sokáig kell várni, ez megakadályozza a külső randevú befejeződését (*rejtett várakozás*). A belső randevú mindig feltételes legyen!

15.8.2. A select utasítás hívó taszkban

A feltételes belépési pont hívás

Az eddigi fejezetekben azt mutattuk be, hogy egy taszk hogyan tud szelektív módon válaszolni a belépési pont hívásokra.

Ha egy belépési pont hívás-fogadására nem akarunk tetszőlegesen hosszú ideig várakozni, akkor *feltételes* vagy *időhöz kötött belépési pont hívást* alkalmazunk. Feltételes belépési pont hívás ciklusba szervezése nem kívánatos hatékonysági okokból. Egyszerre egynél több belépési pont hívása nem megengedett, helyette *lekérdezési technika* alkalmazható.

A feltételes belépési pont hívás esetén a select utasításnak van egy belépési pont hívást tartalmazó ága és egy else ága. Ebben az esetben belépési pont hívás csak akkor történik, ha a randevú azonnal létre is jön. Nem valósul meg a belépési pont hívás, ha a randevú nem jöhet azonnal létre (pl. mert a taszk végrehajtása még nem tart az accept utasításnál, vagy az accept ága nem nyitott, vagy az illető belépési ponthoz tartozó várakozási sor nem üres). Ha a hívás nem történik meg, helyette az else ág kerül végrehajtásra.

```
procedure Próba(R: in Erôforrás) is
begin
   loop
      select
         R.Lefoglal;
      else
         null; -- várakozás Foglalt = True -ra
      end select;
   end loop;
end Próba;
```

Időhöz kötött belépési pont hívás

Az időhöz kötött belépési pont hívásnál a select utasításnak két alternatívája van: egy belépési pont hívást és egy delay utasítást tartalmazó

ág. A hívás akkor következik be, ha a randevú még a `delay` utasításban megadott idő lejárta előtt létrejön. Ha az adott idő alatt a randevú nem jön létre, akkor a `delay` - ág hajtódik végre.

```
select
   Hívás(Paraméter);
or
   delay 4.0;
end select;
```

15.9. Kivételkezelés

A kivételes esetek továbbterjedésének szabályai:

- Ha hiba keletkezik taszk indításakor, akkor `Tasking_Error` váltódik ki a szülőben.
- Ha a belépési pont hívás abnormális, terminált vagy komplett taszkra történik, akkor `Tasking_Error` keletkezik a hívóban.
- Egy kivételes eset egy randevú belsejében, amelyet nem kezeltek le a randevú végéig kivételes eset lesz a hívóban is és a fogadóban is.

15.10. Példák

Kölcsönös kizárás

Párhuzamos rendszerekben szükség van olyan eszközre, ami biztosítja, hogy amikor egy közösen használt objektumhoz egy taszk „hozzányúl", arra az időre a többi taszk ne férhessen hozzá. Az Adában a kölcsönös kizárást többféleképpen, és nagyon egyszerűen meg lehet valósítani.

Először az osztott objektumot magában a taszkban helyezzük el. Itt `T` egy korábban definiált típus, `Osztott` pedig az osztott objektum neve.

15.7. Példa

```
task Osztott_Változó is
   entry Módosít(X: in T);
   entry Olvas(X: out T);
end Osztott_Változó;

task body Osztott_Változó is
   Osztott: T;
begin
```

```
   loop
      select
         accept Módosít(X: in T) do
            Osztott := X;
         end Módosít;
      or
         accept Olvas(X: out T) do
            X := Osztott;
         end Olvas;
      or
         terminate;
      end select;
   end loop;
end Osztott_Változó;
```

Az osztott objektum lehet globális objektum is, ilyenkor a kölcsönös kizárást szemafor használatával valósíthatjuk meg. Egy szemaforhoz egy `P` és egy `V` művelet tartozik. A `P` művelet csak az első őt meghívó taszkot engedi tovább, a többit várakoztatja, a `V` művelet pedig mindig továbbindít egy `P` által felfüggesztett taszkot.

15.8. Példa

```
task type Szemafor is
   entry P;
   entry V;
end Szemafor;

task body Szemafor is
begin
   loop
      accept P;
      accept V;
   end loop;
end Szemafor;
```

Az osztott objektum használata előtt tehát mindig meg kell hívni a `P` műveletet, amikor pedig befejeződtek az osztott objektumon végzett műveletek, akkor meg kell hívni a `P` műveletet, ezzel biztosítottuk az objektum használatára a kölcsönös kizárást.

```
Osztott: T;
   S: Szemafor;
begin
```

```
S.P;
Osztott := F(Osztott);
S.V;
...
```

A szemafor nem túlságosan megbízható szinkronizációs technika, mert ha valahol kifelejtjük a V művelet meghívását, az teljesen összezavarja a program működését.

Termelő-fogyasztó feladat

A termelő-fogyasztó feladatokban kétfajta taszk szerepel: az egyik, ami bizonyos típusú adatokat előállít, a másik pedig az, amelyik feldolgozza ezeket.

15.9. Példa

```
with Osztott_Sor;
   package Int_Osztott_Sor is new Osztott_Sor(Integer, 128);
task body Termelô is
   C: Integer;
begin
   loop
      -- C elöállítása
      Int_Osztott_Sor.Sor.Betesz(C);
   end loop;
end Termelô;

task body Fogyasztó is
   C: Integer;
begin
   loop
      Int_Osztott_Sor.Sor.Kivesz(C);
      -- C felhasználása
   end loop;
end Fogyasztó;
```

A még fel nem dolgozott adatokat egy sorban kell tárolni, s ehhez a sorhoz mind a két fajta taszknak hozzá kell tudni férni, sőt, a hozzáférésre biztosítani kell a kölcsönös kizárást. Egy ilyen sor kezelését mutatjuk meg az alábbi csomagban.

15.10. Példa

```
generic
   type Elem is private;
   Sor_Hosszúság: Natural;
```

```
package Osztott_Sor is

   task Sor is
      entry Kivesz (D:out  Elem);
      entry Betesz (D: in Elem);
   end Sor;
end Osztott_Sor;

package body Osztott_Sor is
   task body Sor is
      type P is new Natural range 0 .. Sor_Hosszúság;
      subtype Aktuális_Tároló is P range 1 .. P'Last;
      Tároló: array (Aktuális_Tároló) of Elem;
      Benne: P := 0;
      Inx, Outx: Aktuális_Tároló:= 1;
   begin
      loop
         select
            when Benne > 0 =>
            accept Kivesz (D: out Elem) do
               D := Tároló(Outx);
            end Kivesz;
            Outx := Outx mod P'Last + 1;
            Benne := Benne - 1;
         or
            when Benne < P'Last =>
            accept Betesz (D: in Elem) do
               Tároló(Inx) := D;
            end Betesz;
            Inx := Inx mod P'Last + 1;
            Benne := Benne + 1;
         or
            terminate;
         end select;
      end loop;
   end Sor;
end Osztott_Sor;
```

Osztott halmaz megvalósítása

A következő példa olyan halmazt valósít meg, melynek elemeit az `Id` diszkrét típus értékeivel mint kulcsokkal azonosítjuk. Ezekhez a kulcsokhoz Duration értékkel megadott időtartamok tartoznak. A halmaz osztott, több taszk is használhatja.

15.11. Példa

```
type Id is range 1 .. 128;
package Halmaz is
   Üres: exception;
   procedure Beszúr(J: in Id; T: in Duration);
   procedure Legkisebb(J: out Id);
end Halmaz;

package body Halmaz is
   task Vezetô is
      entry Beszúr (J: in Id; T: in Duration);
      entry Legkisebb (J: out Id; P: out Boolean);
   end Vezetô;

   task body Vezetô is
      H: array (Id) of Boolean := (Id => False);
      R: array (Id) of Duration;
   begin
   loop
      select
         accept Beszúr  (J: in Id; T:in Duration) do
            R(J) := T;
            H(J) := True;
         end Beszúr;
      or
         accept Legkisebb (J: out Id; P: out Boolean) do
            declare
            T: Duration := Duration'Last;
            S: Id;
            Talál: Boolean := False;
            begin
               for I in Id loop
                  if H(I) and then R(I) <= T then
                  S := I; T := R(I); Talál := True;
                  end if;
               end loop;
               if Talál then
                  H(S) := False;
                  J := S;
                  P := True;
               else
                  P := False;
               end if;
            end;
```

```
         end Legkisebb;
      or
         terminate;
      end select;
   end loop;
   end Vezetô;

   procedure Beszúr(J:in Id; T:in Duration) is
   begin
      Vezetô.Beszúr(J, T);
   end Beszúr;

   procedure Legkisebb(J:out Id) is
      P: Boolean;
   begin
      Vezetô.Legkisebb(J, P);
      if not P then raise Üres; end if;
   end Legkisebb;
end Halmaz;
```

Forgalmi lámpák irányítása

Egy kereszteződés forgalmi lámpáinak vezérlését végző programot készítünk. A kereszteződésben egy főút és egy mellékút metszi egymást. Általában a főúton szabad a jelzés, és a főút forgalmát csak akkor állítjuk meg, ha a mellékútba beépített érzékelő azt jelzi, hogy a mellékúton a kereszteződéshez gépkocsi érkezett, vagy ha egy gyalogos azt jelzi, hogy a főúton levő zebrán át akar haladni. Az érzékelő és a gyalogos nyomógombja egyaránt a 8#2000# címen okoznak megszakítást. A főút forgalmának megállításakor be kell tartani a következőket:

- A főút forgalmát csak akkor lehet megállítani, ha legalább 3 percen keresztül a zöld jelzés volt érvényben.
- A főút forgalmát legfeljebb fél percre szabad leállítani.

A forgalmi lámpák kapcsolására a `Forgalmi_Lámpa` csomagot használhatjuk:

```
package Forgalmi_Lámpa is
procedure Bekapcsol;
   -- a lámpák bekapcsolása; a fôút lámpája zöld,
   -- a mellékúté piros
procedure Vált;
   -- ellenkezôre állítja a kereszteződésben a lámpákat
end Forgalmi_Lámpa;
```

A főprogram:

15.12. Példa

```
with Forgalmi_Lámpa; use Forgalmi_Lámpa;
procedure Lámpa is
task Világít is
   entry Stop;
   for Stop use at 8#2000#;
end Világít;

task body Világít is
   Kikapcsol: constant Duration := 180.0;
   Nyitva: constant Duration := 30.0;
begin
   Bekapcsol;
   loop
      delay Kikapcsol;
      accept Stop; Vált;
      delay Nyitva;
      for I in 1 .. Stop'Count loop
         accept Stop;
      end loop;
      Vált;
    end loop;
end Világít;
begin
   null;
end Lámpa;
```

Prímszámok előállítása

Prímszámok előállításának egy jól ismert módszere az Eratoszthenesz - féle szitamódszer. A taszkok felhasználásával érdekes program készíthető az alábbiak alapján: a természetes számokat sorba állítjuk, majd az első prímszámmal elindítunk egy szűrő taszkot. A szűrő taszk az első kapott számot kiírja, hiszen ez biztosan prím, elindít egy újabb szűrő taszkot, s a számokat – kivéve az első szám többszöröseit – ennek átadja.

15.13. Példa

```
with Text_io; use Text_io;
procedure Rosta is

package Int_Io is new Integer_io(Integer); use Int_Io;
```

```
-- Definiáljuk a Szürö típust
task type Szûrô is
   entry Sor(Szám: in Integer);
end Szûrô;

type Szûrô_Task is access Szûrô; -- mutató a Szürô taszkra
Elsô_Szûrô: Szûrô_Task;
Számok: Integer;

function Új_Szûrô return Szûrô_Task is
begin
   return new Szûrô;
end Új_Szûrô;

task body Szûrô is
   Köv_Szûrô: Szûrô_Task;
   Prím, N: Integer;
begin
   accept Sor(Szám: in Integer) do
      Prím := Szám;
   end Sor;
   if Prím > -1 then -- "-1" az utolsó számot jelöli
      Put(Prím); New_Line;
      Köv_Szûrô := Új_Szûrô; -- új szüröt hoz létre
      loop
         accept Sor(Szám: in Integer) do
            N := Szám;
         end Sor;
         if N = -1 then
            Köv_Szûrô.Sor(N);
            exit;
         end if;
         if N mod Prím /= 0 then
            -- továbbadja a számot
            Köv_Szûrô.Sor(N);
         end if;
      end loop;
   end if;
end Szûrô;
begin
   Put("Add meg az intervallumot: ");
   Get(Számok);
   Elsô_Szûrô := new Szûrô;
   Put_Line("A prímszámok: ");
```

```
   for I in 2 .. Számok loop
      Elsô_Szûrô.Sor(I);
   end loop;
   -- az utolsó szám jelölése
   Elsô_Szûrô.Sor(-1);
end Rosta;
```

15.11. Feladatok

15.1. Oldjuk meg a QuickSort feladatot taszkok segítségével.

15.2. Egy kereszteződésbe a különféle irányokból adott gyakorisággal érkeznek a gépkocsik. A kereszteződésben elhelyezett forgalmi lámpa adott szabályok szerint működik. Készítsünk taszkok felhasználásával egy programot az előálló forgalmi dugók vizsgálatára.

15.3. Készítsünk taszkok segítségével egy programot, amely egy lift működését szimulálja.

15.4. Oldjuk meg az étkező filozófusok problémát taszkok segítségével.

16. Fejezet

Védett típusok és objektumok

Az Ada taszkok sok feladatra igen kényelmesen használhatók. Vannak azonban a párhuzamosság témakörén belül olyan feladatok is, amelyek megoldásához más eszközre lehet szükség. Nézzünk erre egy példát. Tegyük fel, hogy van egy `Kiír` nevű eljárásunk, amit több taszk is használ. Azt szeretnénk biztosítani, hogy egyszerre legfeljebb egy olyan taszk lehessen, ami éppen végrehajtja `Kiír`-t. E nélkül ugyanis elképzelhető lenne az, hogy a különböző taszkok által kiírt dolgok összekeverednének a kimeneten:

```
Én vagyok azÉn vagyok a másik egyik taszk.
taszk.
```

Ha a `Kiír` műveletet nem eljárásként, hanem egy taszk belépési pontjaként valósítjuk meg, a végrehajtása egy időben csak egy taszkban történhet.

16.1. Példa

```
task Védett is
  entry Kiír ( ... );
end Védett;

task body Védett is
begin
  loop
    select
       accept Kiír ( ... ) do ... end;
    or terminate;
    end select;
  end loop;
end Védett;
```

A végrehajtást a `Védett.Kiír(...)` hívással lehet kezdeményezni. Mivel a `Védett` taszk egyszerre csak egy másik taszkkal lehet randevúban, a `Kiír` törzse egyszerre csak egy példányban futhat.

Ugyanezt a hatást érhetjük el *védett objektumok* használatával is. Segítségükkel azonban sokkal elegánsabb – és ami még fontosabb, hatékonyabb[1] – megoldáshoz juthatunk.

16.2. Példa
```
protected Védett is
  procedure Kiír ( ... );
end Védett;

protected body Védett is
  procedure Kiír ( ... ) is ... end;
end Védett;
```

Egy védett objektum tartalmazhat adattagokat, valamint ezen adattagok elérését biztosító műveleteket. Ezekre a műveletekre párhuzamos környezetben automatikusan kölcsönös kizárás teljesül. Védett objektumokkal bonyolult *szinkronizációs szerkezeteket* építhetünk fel. Segítségükkel sok olyan feladatot megoldhatunk, amelyek megoldása taszkokkal kényelmetlenebb, kevésbé hatékony lenne.[2] Ennek elsősorban az az oka, hogy a taszkok közötti szinkronizációs mechanizmus, a randevú, vezérlés-centrikus, ezzel szemben a védett objektumok használata *adat-centrikus* szinkronizációt tesz lehetővé. Másrészről a védett objektumok viszonylag könnyen fordíthatóak nagyon hatékony tárgykóddá. A taszkokkal ellentétben a védett objektumok *passzív szerkezetek*, megvalósításukhoz nincs szükség saját processzorra, illetve időosztásos működtetés esetén kontextus-váltásra.

16.1. Áttekintés

A védett objektumok tartalmazhatnak adattagokat és műveleteket. Az adattagokhoz a védett objektumon kívül nem lehet hozzáférni, csak a műveleteken keresztül. A műveleteket három csoportba oszthatjuk: lehetnek függvények, eljárások és belépési pontok. A védett objektum függvényeire és eljárásaira a *védett függvény*, *védett eljárás*, *védett alprogram* elnevezéseket használhatjuk.

A védett objektum célja az, hogy párhuzamos környezetben a műveletek végrehajtását (és így egyben az adattagokhoz való hozzáférést is) szinkronizálja. Az eljárásokra és a belépési pontokhoz rendelt törzsekre *kölcsönös kizárást* biztosít, a függvények esetén azonban nem tiltja a *konkurens végrehajtást*. A függvények nem változtatják meg a védett objektum állapotát,[3]

[1] A hatékonysági kérdésekre visszatérünk a 16.3.1. szakaszban.

[2] Az Ada83-ban nem voltak még védett objektumok. Bevezetésük fontos, a nyelv felhasználói által is sürgetett újítás volt az Ada95-ben.

[3] Bizonyos – extrém – esetekben egy függvény hívása is megváltoztathatja a védett objektum állapotát. Részletekért lásd 339. o.

csak lekérdezik (olvassák) azt: ezért semmi baj nem történhet, ha egy időben több függvény (vagy egy függvény több példányának) végrehajtása történik. Ezzel szemben az eljárások és a belépési pontokhoz rendelt törzsek megváltoztathatják (írhatják) a védett objektum adattagjait. Ha megengednénk, hogy egy eljárással vagy egy belépési pont törzzsel egyidőben egy másik művelet is fusson, azt kockáztatnánk, hogy a védett objektum inkonzisztens állapotba kerül.

16.3. Példa

```
protected Változó is
  function Lekérdez return Adat;
  procedure Beállít ( Új: in Adat );
private
  V: Adat;
end Változó;

protected body Változó is
  function Lekérdez return Adat is
  begin
    return V;
  end Lekérdez;

  procedure Beállít ( Új: in Adat ) is
  begin
    V := Új;
  end Beállít;
end Változó;
```

Ebben az egyszerű példában a `Beállít` eljárás végrehajtása közben sem másik `Beállít` példány, sem `Lekérdez` nem futhat. Ha ugyanis a `Beállít` a `V` új értékét félig már beállította volna, és ekkor mondjuk a `Lekérdez` olvasná a `V`-t, már nem a régi, de még nem is az új, hanem esetleg egy teljesen értelmetlen értéket látna.[4]

A *belépési pontok* abban különböznek az eljárásoktól, hogy tartozik hozzájuk egy *várakozási sor*, és hogy egy *őrfeltételt* kell hozzájuk rendelni. Ha az őrfeltétel híváskor igaz, a belépési ponthoz tartozó törzs végrehajtódik; ha nem, a belépési pontot hívó taszk bekerül a belépési ponthoz tartozó várakozási sorba. Amikor az őrfeltétel igazzá válik, egy taszk kikerül a várakozási sorból és végrehajtja a belépési ponthoz rendelt törzset.

[4] Az olvasók-és-írók probléma (readers-writers problem) egy híres, alapvető szinkronizációs feladat: az író folyamatok végrehajtása alatt sem írás, sem olvasás nem történhet, de az olvasó folyamatok futhatnak egyidőben. Vegyük észre, hogy a védett objektumok azzal, hogy megkülönböztetik a függvényeket az eljárásoktól és a belépési pontoktól, automatikusan megoldják ezt a feladatot.

16.4. Példa

```
protected Osztott_Verem is
  procedure Betesz ( E: in Elem );
  entry Kivesz ( E: out Elem );
private
  V: Verem;
end Osztott_Verem;

protected body Osztott_Verem is
  procedure Betesz ( E: in Elem ) is
  begin
    Betesz(V,E);
  end;

  entry Kivesz ( E: out Elem ) when Méret(V) > 0 =>
  begin
    Kivesz(V,E);
  end;
end Osztott_Verem;
```

Az őrfeltételek újra kiértékelése olyankor történik, amikor a védett objektum egy eljárása vagy egy belépési pontjához rendelt törzs véget ér. Ezek a műveletek ugyanis megváltoztathatják a védett objektum állapotát, így az őrfeltétel értékét is. A védett függvények – elvárásaink szerint – nem módosítják a védett objektumot,[5] így végrehajtásuk után az őrfeltételek nem értékelődnek ki újra.

Bemutatjuk ezeket a szabályokat egy lehetséges eseménysorozaton. Tegyük fel, hogy először is egy taszk meghívja a `Kivesz` belépési pontot:

```
Osztott_Verem.Kivesz(X);
```

Kiértékelődik az őrfeltétel, és hamisnak találtatik. A taszk bekerül a `Kivesz` belépési pont várakozási sorába. Ugyanez történik minden további olyan taszkkal, amelyik a `Kivesz`-t hívja. Más a helyzet a `Betesz` eljárással. Az alprogramokhoz nem tartozik őrfeltétel: ha egy taszk meghívja `Betesz`-t,

```
Osztott_Verem.Betesz(42);  -- tfh. az Elem itt Integer
```

az eljárás – természetesen a kölcsönös kizárás szabályait betartva – azonnal végrehajtódhat. Miután a törzs véget ért (és mivel nem függvényről volt szó), a védett objektum ellenőrzi, hogy a `Kivesz` belépési pont várakozási sora üres-e. Ha nem, kiértékeli az őrfeltételt. Ebben az esetben az eredmény igaz lesz, tehát az első várakozó taszkot „beengedi": a belépési ponthoz rendelt törzs végrehajtódhat. Ezután (mivel megint csak nem egy függvény

[5] A programozó felelőssége a függvényeket így elkészíteni. Lásd még: 339. o.

ért véget,) ellenőrzi a várakozási sort. Ha az még mindig nem üres, kiértékeli az őrfeltételt, ami most hamis lesz: a várakozóknak tovább kell várniuk.

Ha a `Betesz` törzsének végrehajtása után a `Kivesz` várakozási sora üres, a következő `Kivesz` hívás igaz őrfeltételt fog találni. Ebben az esetben azonnal végrehajthatja a belépési pont törzsét, nem kerül be a várakozási sorba.

A védett objektumok működését a későbbiekben részletesebben is tárgyalni fogjuk.

16.2. Védett egységek készítése

A védett egységek tartalmazhatnak *adattagokat* és *műveleteket*. Az adattagok változódeklarációk, a műveletek függvények, eljárások és belépési pontok lehetnek. A kommunikáció a műveletek paraméterein keresztül történik, az alprogramoknál és a taszk belépési pontoknál megismert szabályok szerint. Például:

```
Változó.Beállít(Változó.Lekérdez+1);
```

Mint látjuk, a műveletekre kívülről minősített névvel kell hivatkozni, úgy, mint a taszkok belépési pontjaira, vagy mint a `use` nélkül használt csomagok komponenseire. A fenti példákból az is kiderül, hogy a védett objektumon belül a minősítésre nincs szükség.

Védett egység definiálhat *belépésipont-családot* is, a taszkoknál megismert módon. Álljon itt egy példa arra, hogy a belépésipont-családok hogyan használhatók prioritások kezelésére. A `Kijelzô` mindig a legfontosabb üzenetet írja ki:

16.5. Példa

```
type Prioritás is (Magas, Közepes, Alacsony);
protected Kijelzô is
  entry Üzenet (Prioritás) ( Szöveg: in String );
end Kijelzô;

protected body Kijelzô is
  function Mehet ( Aktuális: Prioritás ) return Boolean is
    Fontosabb: Prioritás := Prioritás'Last;
  begin
    while Fontosabb > Aktuális loop
      if Üzenet(Fontosabb)'Count>0 then return False; end if;
      Fontosabb := Prioritás'Pred(Fontosabb);
    end loop;
    return True;
  end Mehet;
```

```
  entry Üzenet (for Aktuális in Prioritás) (Szöveg: in String)
  when Mehet(Aktuális) is
  begin
    Put_Line(Prioritás'Image(Aktuális) & Ascii.HT & Szöveg);
  end Üzenet;
end Kijelzô;
```

A taszkok az alábbi módon küldhetnek üzenetet a kijelzőre:

```
Kijelzô.Üzenet(Magas)("Nekem van igazam!");
```

A belépésipont-család minden tagjához külön várakozási sor tartozik.

16.2.1. Típus és objektum

Ugyanúgy, mint ahogy lehet taszk objektumokat és taszk típusokat írni, készíthetünk *védett objektumokat* és *védett típusokat*. A védett típusokból statikusan (változó-deklarációval) és dinamikusan (allokátorral) is létrehozhatunk védett objektumokat. A védett típusok megadása szintaktikusan abban különbözik a védett objektumokétól, hogy a `protected` kulcsszó után szerepel a `type` kulcsszó is. Az előző példánál maradva:

16.6. Példa

```
protected type Osztott_Verem_Típus is
  procedure Betesz ( E: in Elem );
  entry Kivesz ( E: out Elem );
private
  V: Verem;
end Osztott_Verem_Típus;
```

A törzs (a néven kívül) nem változik. Statikusan így hozhatunk létre védett objektumot ebből a típusból:

```
Osztott_Verem_Változó: Osztott_Verem_Típus;
```

A `new` allokátor használatával pedig dinamikusan kreálhatunk védett objektumot egy megfelelő mutató típusból:

```
type Osztott_Verem_Mutató_Típus is access Osztott_Verem_Típus;
Osztott_Verem_Mutató: Osztott_Verem_Mutató_Típus;
...
Osztott_Verem_Mutató := new Osztott_Verem_Típus;
```

Az `Osztott_Verem` deklarációját *egyedüli védett objektum* deklarációnak, az `Osztott_Verem_Típus` deklarációját védett típus deklarációnak fogjuk nevezni; összefoglaló néven a kettőt *védett egység* deklarációjának hívjuk.

16.2.2. Szerkezet

A védett egységek ugyanúgy épülnek fel, mint a csomagok és a taszkok: van egy specifikációs részük és egy törzsük. A deklarációs részük szintén tartalmazhat *átlátszatlan részt*, amit a `private` kulcsszó vezet be. A védett egység interfészét a deklaráció látható része tartalmazza. Ez alprogram- és belépési pont-deklarációkból állhat. Műveleteket deklarálhatunk az átlátszatlan részben, sőt, alprogramokat a törzsben is. A csomagoknál megismert szabályok szerint a törzs tartalmazza a deklarált műveletek megvalósítását.

Adattagok csak a védett egység deklarációjának átlátszatlan részében szerepelhetnek, a védett objektumon kívülről csak annak látható műveleteivel érhetők el. Ez a szabály az azonos típusba tartozó különböző védett objektumokra is vonatkozik: az egyik védett objektum nem hivatkozhat a másik adattagjaira. Nem zárja ki viszont a nyelv azt a lehetőséget, hogy a védett egység műveletei a védett objektumon kívüli, globális változókra hivatkozzanak. Az sem szabálytalan, ha a védett objektum egy adattagja mutató típusú, és a mutatott objektum a program más egységei számára is hozzáférhető (akár deklaráció, akár egy másik mutató segítségével).

16.7. Példa

```
N, M: aliased Integer := 1;
type Integer_Mutató is access all Integer;
protected Veszélyes is
  function Érték return Integer;
private
  Változó: Integer_Mutató := N'Access;
end Veszélyes;

protected body Veszélyes is
  function Érték return Integer is
  begin
    N := N+1;
    return Változó.all + M;
  end Érték;
end Veszélyes;
...
N := 2; -- megváltoztatja Veszélyes.Érték eredményét
M := 2; -- ez szintén
```

A nyelv rugalmasságának megtartása (és implementálhatósága) érdekében helyesnek kell tekintenünk olyan programokat is, amelyek egy védett objektum állapotát a műveletei megkerülésével, ezeken a globális változókon keresztül változtatják meg. A függvények mellékhatás-mentességét sem követeli (nem is követelheti) meg az Ada: figyeljük csak meg a `N := N+1;` értékadást a `Veszélyes.Érték` függvény belsejében... A programozó fele-

lőssége az, hogy az ilyen, indirekt módon véghezvitt állapotváltoztatásokból baj ne származzon.

Fontos, hogy a védett egység törzse sem adattagot, sem belépési pontot nem deklarálhat. Ennek az az előnye, hogy védett objektum létrehozásánál fordítási időben le lehet foglalni az objektum számára szükséges tárterületet. A belépési pontok ebből a szempontból azért esnek az adattagokkal azonos elbírálás alá, mert a védett objektum létrehozásakor allokálni kell a hozzájuk tartozó várakozási sor számára szükséges helyet is.

A védett egységek deklarációja és törzse is tartalmazhat reprezentációs záradékokat. (Lásd 8.3.) Az adattagokra, a védett egység helyfoglalására azonban reprezentációs záradék nem írható: a belépési pontokhoz tartozó várakozási sorok a reprezentáció közvetlen megadását lehetetlenné teszik.

16.2.3. Diszkrimináns

A védett típusok deklarációjában szerepelhetnek diszkriminánsok is. A lehetőségek ugyanazok, mint a taszkok és a rekordok esetén: diszkrimináns diszkrét típus és mutató definíció lehet. Ebben a példában a `Szemafor` típusnak egy diszkriminánsa van:

16.8. Példa

```
protected type Szemafor ( Max: Positive := 1 ) is
  entry P;
  procedure V;
private
  Bent: Natural := 0;
end Szemafor;

protected body Szemafor is
  entry P when Bent < Max is
  begin
    Bent := Bent+1;
  end P;

  procedure V is
  begin
    Bent := Bent-1;
  end V;
end Szemafor;
```

A szemafor egy ún. kritikus szakaszt őriz. Ezen a szakaszon belül egyszerre legfeljebb `Max` folyamat tartózkodhat. A belépés előtt a `P`, a kilépés után

a `V` műveletet kell a folyamatoknak meghívni.[6] Látható, hogy a kilépéshez nincs feltétel szabva, de a belépéshez van.

A diszkriminánsokat használhatjuk arra, hogy az azonos típusba tartozó védett objektumokat felparaméterezzük. A diszkrimináns értéke gyakran szolgál az objektum azonosítására, belépésipont-családok méretének definiálására, de látunk majd egy példát arra is, hogy egy védett objektumot egy rajta kívüli objektummal (pl. megszakítással, 16.11) a diszkriminánsán keresztül kapcsolunk össze.

16.2.4. Őrfeltételek

A belépési ponthoz tartozó törzs csak akkor hajtódhat végre, ha teljesül a belépési ponthoz rendelt őrfeltétel. Noha sok szempontból hasonlít ez az őrfeltétel a taszkok törzsében alkalmazott `select` utasítás ágaihoz rendelhető őrfeltételhez, van egy fontos különbség: a kiértékelési stratégia. Egy belépési pont meghívásakor, amikor a kölcsönös kizárást biztosító mechanizmus engedélyezi a belépési pont törzsének végrehajtását, kiértékelődik az őrfeltétel. Ha igaz, a törzs végrehajtása folytatódhat, ha nem, a hívó taszk felfüggesztődik és bekerül a belépési pont várakozási sorába. A nem üres várakozási sorral rendelkező belépési pontok őrfeltétele akkor értékelődik ki újra, amikor befejeződik a szóban forgó védett objektum egy eljárásának vagy belépési pont törzsének egy kívülről kezdeményezett végrehajtása.

Az őrfeltétel értéke természetesen máskor is megváltozhat: semmi sem zárja ki például, hogy az őrfeltételben a védett objektumon kívüli változóra legyen hivatkozás. Egy ilyen változót a program a védett objektum „tudta nélkül" is meg tud változtatni, és ez az őrfeltételre is hatással lehet. Nézzük például a `Veszélyes` objektumot. Ha egy belépési pont őrfeltételében szerepel a `Érték` függvény, egy globális változó értékének módosítása, vagy akár az `Érték` függvény meghívása is megváltoztathatja az őrfeltételt. Ennek ellenére az nem fog újra kiértékelődni. Igyekezzünk programjainkban kerülni az ilyen helyzeteket, de ha ez nem lehetséges, alaposan gondoljuk át, melyik utasítás mire van hatással.

Az őrfeltételben nem csak a védett objektum komponenseire (adattagjaira, függvényeire, diszkriminánsaira) és globális egyedekre hivatkozhatunk: szerepelhet bennük belépésipont-család indexe is. Erre láthattunk példát a `Kijelző` objektum definiálásánál. (337 o.) Nem szerepelhetnek azonban az őrfeltételben a belépési pont (formális) paraméterei.[7]

[6] A szemafor fogalmát Dijkstra vezette be. Azóta hagyományosan `P`-nek és `V`-nek nevezik a két műveletét. A nevek a holland passeren (áthalad) és vrijmaken (szabaddá tesz) szavak kezdőbetűiből származnak.

[7] Enélkül a szabály nélkül a belépési pont őrfeltételek újra kiértékelésekor minden várakozó taszkra ki kellene azt értékelni, és a várakozók nem egy sorban, hanem egy halmaz jellegű adatszerkezetben kellene, hogy várakozzanak.

16.2.5. A műveletek belseje

A védett egységek műveleteit az alprogramoknál és a taszk belépési pontoknál megszokott módon készíthetjük el. Definiálhatunk a védett objektumon kívülről nem hívható, lokális műveleteket is. (A lokális belépési pontokat a deklaráció átlátszatlan részében deklarálni kell!) A védett objektum műveletei meghívhatják egymást – mint azt a `Kijelzô`-s példában láttuk –, és persze a védett objektumra nézve globális műveleteket (alprogramokat, vagy akár belépési pontokat) is.

A védett műveletekre vonatkozó szabály az, hogy a programozónak igyekeznie kell úgy megírni azokat, hogy végrehajtásuk minél rövidebb, vagy legalábbis korlátos ideig tartson. Arra mindenféleképpen illik vigyázni, hogy nehogy a műveletek végessége sérüljön: ezt a veszélyt hordozzák ugyanis magukban az ún. *potenciálisan blokkoló* műveletek.

Az Ada nem tiltja, de hibásnak tartja, ha egy védett művelet egy potenciálisan blokkoló műveletet meghív. Mik ezek a műveletek? Azok, amelyek végrehajtása egy feltétel igazzá válásáig, egy esemény bekövetkezéséig felfüggesztődhet. Ilyen például egy belépési pont hívása, vagy taszkok esetén egy `accept` utasítás. Könnyen előfordulhat ugyanis, hogy a blokkoló utasítások egymás befejeződésére várnak, a végtelenségig. Az ilyen helyzetet *holtpontnak* szokás nevezni.

Nézzünk egy példát, ami ugyan túl sok gyakorlati jelentőséggel nem bír, de a probléma szemléltetésére megfelel. A védett objektum faktoriálist számol, rekurzióval. Figyeljük meg, hogy a `Faktoriális_Számolás` eljárás önmagát hívja. Ez egy úgynevezett belső hívás, az ilyenre a védett objektum nem biztosít kölcsönös kizárást.

16.9. Példa

```
protected type V_Típus is
  procedure Faktoriálist_Számol ( X: in out Natural );
end;

protected body V_Típus is
  procedure Faktoriálist_Számol ( X: in out Natural ) is
    Y: Natural;
  begin
    if X = 0 then X := 1;
    else Y := X-1; Faktoriálist_Számol(Y); X := X*Y;
    end if;
  end;
end;
```

Egy `V_Típus` típusú objektum `Faktoriálist_Számol` műveletének meghívása jól fog működni, a paraméterben megadott változóba belemásolja a kez-

deti érték faktoriálisát. A program ugyanúgy működik, ha a védett eljárás törzsében a rekurziót kiváltó `Faktoriálist_Számol(Y)` hívás helyett a

```
V_Típus.Faktoriálist_Számol(Y);
```

sort írjuk. A védett egységeken belül, akár védett típusról, akár egyedüli védett objektumról van szó, a védett egység neve az aktuális objektumot jelenti. (Pontosan ugyanúgy, mint a taszkoknál.[8]) Ez a kifejezés szintén belső hívásnak számít. Más a helyzet azonban, ha az előbbi kódrészletet kiegészítjük az alábbiakkal:

```
W: V_Típus;

procedure Külsô ( X: in out Natural ) is
begin
  W.Faktoriálist_Számol(X);
end;
```

Cseréljük ki a `V_Típus.Faktoriálist_Számol(Y)` belső hívást a

```
Külsô(Y);
```

sorra. Tegyük fel, hogy egy taszk meghívja a `W.Faktoriálist_Számol` eljárást. A rekurzió helyén a `Külsô` eljáráson keresztül ismét saját magát fogja hívni ez a védett eljárás, de ebben az esetben a hívás már külsőnek (és így potenciálisan blokkolónak) minősül: a `W` védett objektum kölcsönös kizárást biztosít rá. Ezért a második hívás nem tud végrehajtódni, amíg az első, ami őt meghívta, véget nem ér, ami szintén blokkolt állapotban van, amíg a második be nem fejeződik. Holtpont alakult ki.[9]

A lokális belépési pontok és a requeue utasítás

A védett egységekben definiálhatunk olyan belépési pontokat, amelyek a védett objektumon kívülről közvetlenül nem hívhatók meg. Ezeket a belépési pontokat a védett egység deklarációjának átlátszatlan részében deklaráljuk. Használatuk csak a védett objektumon belülről lehetséges. Mivel a belépési pont hívása blokkoló művelet, meghívásuk holtpontot eredményez,

[8] Érdekes a sablon alprogramokkal is párhuzamot vonni: egy sablon alprogram törzsében a sablon alprogram neve nem a sablon alprogramot, hanem az aktuális példányt jelöli, lásd 11.1.

[9] Az, hogy az Ada nem tiltja, de hibásnak tartja a potenciálisan blokkoló műveletek hívását a védett műveleteket belül, azt jelenti, hogy a program lefordul, de a működése nem definiált: lehet, hogy futási idejű hiba váltódik ki, és az is lehet, hogy holtpont keletkezik. Ha a fenti példában a `Külsô(Y)` helyett `W.Faktoriálist_Számol(Y)`-t írunk, a nyelv definíciója szerint szintén külső (tehát potenciálisan blokkoló) hívást kapunk. Az általunk használt (gnat 3.09) fordító ezt az esetet úgy kezeli, mintha belső hívás történt volna.

így annak igazándiból nem sok értelme van. Ezzel szemben a lokális belépési pontok nagyszerűen használhatók a `requeue` utasítással.[10]

A `requeue` utasítás egy belépési ponthoz rendelt törzsben a belépési pont aktuális hívását elhelyezi az utasítás paramétereként megadott belépési pont várakozási sorában. Ez a második belépési pont lehet ugyanaz a belépési pont, de lehet egy másik is; sőt, lehet akár egy másik védett objektum vagy taszk belépési pontja is. Az egyetlen megkötés, hogy ennek a második belépési pontnak vagy ne legyenek paraméterei, vagy egyezzen meg a paraméterlistája azon belépési pont paraméterlistájával, amelyik törzsében a `requeue` szerepel.

Gyakran előfordul, hogy egy védett objektum által nyújtott szolgáltatást két lépcsőben kell elvégezni. Az első lépcső után a szolgáltatást igénylő (azaz a belépési pontot hívó) taszkot fel kell függeszteni, amíg a második lépcső feltételei meg nem teremtődnek. Ennek implementálásához hasznos, ha egy belépési pont-hívást át lehet mozgatni egy másik belépési pont várakozási sorába.[11]

Példaként vizsgáljuk meg az üzenetszórásos szignál programját. (Lásd 16.4.2.) Ebben a belső `Reset` belépési pontot használjuk a `requeue` utasítás paramétereként.

16.2.6. Működés

Ebben a szakaszban a védett objektumok működését szemléltetjük. Már tudjuk azt, hogy az eljárásokra és a belépési pontokra kölcsönös kizárás teljesül, és azt is, hogy a függvények konkurensen is végrehajthatók. Az e mögött húzódó mechanizmust a *lakat*[12] fogalmával írhatjuk le. Minden védett objektumhoz tartozik két lakat: egy a csak olvasó műveletek (függvények) és egy az író/olvasók (eljárások és belépési pontok) számára. Az író/olvasó lakathoz csak egy *kulcs* tartozik, a csak olvasóhoz viszont korlátlan sok.

Ha egy folyamat végre akar hajtani egy műveletet, lezárja a műveletnek megfelelő lakatot. Ha ez a művelet írhat (eljárás vagy belépési pont), akkor amíg véget nem ér, újabb művelet nem indulhat el; ehhez ugyanis az kellene, hogy az író/olvasó lakaton keresztüljusson. Ez lehetetlen, hiszen csak egy író/olvasó kulcs van, és azt az éppen futó folyamat birtokolja. Ha az éppen futó művelet csak olvasás, más a helyzet. Az írni akaró folyamatok továbbra

[10] Taszkoknál a deklaráció átlátszatlan részében deklarált belépési pontok szintén csak a taszkon belülről hivatkozhatók: a taszkon belül definiált (al)taszkok tudják csak meghívni.

[11] Egy másik megoldás az lenne, hogy a szolgáltatást kérő taszk két belépési pontot hívna meg egymás után. Ez a megoldás egyrészt azért rosszabb, mint az előző, mert nem biztosítja a szolgáltatás atomicitását, ha a szolgáltatás két lépcsője közvetlenül egymás után végrehajtható. Másrészt ez a megoldás a szolgáltatás megszervezésében használt belső protokollt is láthatóvá teszi a szolgáltatás felhasználói számára.

[12] Angolul „lock".

sem juthatnak be, hiszen nem rendelkeznek olvasás kulccsal. Egy második olvasás viszont igen, így az elsővel párhuzamosan végrehajtódhat. Amíg az író/olvasó lakat nincs bezárva, a csak olvasni szándékozó folyamatok mind végrehajtódhatnak.

Ha egy időpillanatban senki sem használja az objektumot, de több folyamat is szeretné, nem definiált, hogy az első kulcsot ki kapja meg. Egy sokprocesszoros rendszeren egy lehetséges megvalósítás az, hogy a folyamatok „aktívan várnak" a belépési jog megszerzésére.[13]

A védelem két szintje

Miután egy védett eljárás vagy egy belépési pont törzs befejeződik, a védett objektum nem üres várakozási sorú belépési pontjainak őrfeltétele újra kiértékelődik. Ha valamelyik közülük igaz, az első, addig abban a sorban várakozó hívás végrehajtódhat. Fontos, hogy az őrfeltételek tesztelése még azelőtt lejátszódik, mielőtt a véget érő író művelet a kulcsát leadta volna, esélyt adva a kulcsra váró folyamatoknak. Úgy is fogalmazhatnánk, hogy a kulcsot először megpróbálja odaadni a várakozási sorokban tartózkodó folyamatok valamelyikének. Ez azt eredményezi, hogy a várakozási sorokban álló taszkok előnyt élveznek azokkal szemben, akik a kulcsra, a „bejutásra" várnak.

Amikor egy taszk meghív egy védett belépési pontot, annak még az őrfeltétele sem értékelődik ki, amíg a taszk meg nem szerzi az írási jogot a védett objektumra. Csak az írási jog megszerzése után dől el, hogy a belépési ponthoz tartozó törzs végrehajtódhat-e, vagy a hívás bekerül a megfelelő várakozási sorba.

A védett objektumokkal mindezek miatt egy kétszintes védelmi rendszert építhetünk fel. Azok a belépési pont-hívások, amelyek már bejutottak az első szinten, de fennakadtak a másodikon (mert az őrfeltételük a kiértékeléskor éppen hamis volt), előnyt élveznek az összes többi védett művelettel szemben. Az üzenetszórásos szignál (lásd 16.4.2.) jól illusztrálja a kétszintes védelmi rendszer előnyeit.

Végre nem hajtott műveletek

Mi történik akkor, ha egy védett objektum működése befejeződik, de még van olyan művelete, amely nem ért véget? Különösnek tűnik ez a kérdés, mert elég nehéz ilyen esetet elképzelni: de nem lehetetlen! Gondoljunk egy idő előtti `Unchecked_Deallocation`-re (lásd 17.5.1.), vagy nézzük az alábbi példát (mint [2]):

[13] Angolul ezt „busy waiting"-nek hívják.

16.10. Példa

```
procedure Main is

  task T is
    entry E;
  end T;

  task body T is

    protected PO is
      entry Ee;
    end PO;

    protected body PO is
      entry Ee when False is
      begin
        null;
      end Ee;
    end PO;

  begin
    accept E do
      requeue PO.Ee;
    end E;
  end T;

begin
  T.E;
end Main;
```

A főprogram meghívja a `T` taszk `E` belépési pontját, de a hívás a **`requeue`** utasítás hatására átkerül a `PO` védett objektum `Ee` belépési pontjának várakozási sorába (ahol a mindig hamis őrfeltétel miatt a végtelenségig kell várnia). Amikor a védett objektum megszűnik, a főprogram hívása még mindig vár.

Az ilyen esetekben az összes várakozó taszkban a hívás helyén kiváltódik egy `Program_Error` kivétel. (Ez összhangban van a taszkoknál megismert mechanizmussal, ahol is egy taszk végetérése az őt hívó taszkokban egy kivétel, mégpedig `Tasking_Error` fellépését eredményezi.)[14]

[14] Az általunk használt (gnat 3.09) fordítóprogram sajnos ezt a példaprogramot nem megfelelően fordítja le.

16.3. A védett egységek használata

Ebben a szakaszban összefoglaljuk a védett egységek használatára, az Ada programokba való beillesztésére vonatkozó szabályokat. Felhívjuk a figyelmet az Ada más nyelvi eszközeivel vonható párhuzamokra is.

Egy védett egység deklarációja jelenthet védett típust vagy egyedüli védett objektumot. A második esetben is definiálódik egy ún. *névtelen típus* (és annak első altípusa): ez lesz az egyedüli védett objektum típusa. Ez a névtelen típus sehogy sem használható újabb objektum létrehozására. Az egyedüli védett objektumokat például nem lehet átnevezni sem (szemben azokkal a védett objektumokkal, amelyek típusa egy védett típus deklarációval deklarált). Ugyanezekkel a szabályokkal találkozhatunk a taszkok esetén is.

A védett típusokat – mint a taszkokat és például a rekordokat – az összetett típusok közé soroljuk. Az említett típusokhoz hasonlóan diszkriminánsokkal is rendelkezhetnek. A diszkrimináns diszkrét vagy mutató típusú lehet.

A védett típusok, mint a taszkok is, eredendően *korlátozott típusok*: sem értékadás, sem predefinit egyenlőségvizsgálat operátor nincs definiálva hozzájuk. Hasonlóan a többi korlátos típushoz, paraméterként hivatkozás szerint adódnak át, és függvény visszatérési értékként is hivatkozás szerintiek.

A védett egységek szerkezete leginkább a taszkokra és a csomagokra emlékeztet a hármas tagolással. Lényeges azonban, hogy egy védett egységen belül nem definiálhatunk típusokat, csomagokat, stb., csak adattagokat és (védett) műveleteket.

A védett egységek a programegységek közé tartoznak (az alprogramok, csomagok, sablonok és taszkok mellett). A védett alprogramok alprogramok, ezért szintén programegységek. A védett egységek nem könyvtári egységek, de a törzsük alegységként (9.3.) külön fordítható. Vegyük például az alábbi két fordítási egységet:[15]

```
procedure P is                -- az egyik fordítási egység
  protected V is end;
  protected body V is separate;
begin
  null;
end;

separate (P) protected body V is end;           -- és a másik
```

[15]Az általunk használt fordító (gnat 3.09) megköveteli, hogy a különböző fordítási egységek különböző fájlokba kerüljenek. Ezt a két fordítási egységet például a `p.adb`, illetve a `p-v.adb` nevű fájlokba kell írni.

Egy védett művelet törzse önmagában nem lehet alegység.

Minden védett művelet (legyen az belépési pont vagy alprogram) kell, hogy rendelkezzen „megvalósítással". Ez a megvalósítás nem lehet átnevezés, csak egy törzs megadása, illetve, ha a nyelv adott implementációja ezt megengedi, `Import` pragma. A másik irányban, egy védett műveletet (alprogramot és belépési pontot, akár egy belépésipont-család egy tagját is) átnevezhetjük, de a belépési pontokat csak eljárásokká.

```
protected V is
  entry E;
  procedure P;
end;

protected body V is
  entry E when True is begin ... end;
  procedure P is begin ... end;
  -- helytelen: procedure PP renames P;
end V;

procedure AP renames V.P;
procedure EP renames V.E;
```

Az alprogramra mutató típusok (3.6.) mutathatnak védett alprogramokra is. Szép példája ennek az `Ada.Interrupts` csomag ezen deklarációja:

```
type Parameterless_Handler is access protected procedure;
```

A védett műveletekre más hívási konvenciók vonatkoznak, mint a közönséges alprogramokra. A belépési pontok hívási konvenciója `entry`, a védett alprogramoké pedig általában `protected`. Ezek a hívási konvenciók biztosítják a védett objektumok eddig megismert műveletvégrehajtási tulajdonságait, a belépési pontok és védett eljárások kölcsönösen kizárt, valamint a védett függvények konkurens végrehajtását. A védett objektumokon belül végrehajtott alprogramhívások (egy lokális alprogram hívása) `Intrinsic` konvenciójú, hiszen ezekre nem érvényesek az előbb említett mechanizmusok. Az `entry` és `protected` szavak csak kulcsszavak, nem használhatók a `Convention` pragmával (lásd 17.6.) hívási konvenció beállítására.

Vannak olyan esetek, amikor ahhoz, hogy egy változóra kölcsönös kizárást biztosítsunk, nincs szükség védett objektumra. Alkalmazhatjuk ugyanis helyette az `Atomic` és a `Volatile` (valamint az `Atomic_Components` és a `Volatile_Components`) pragmákat.[16] (Lásd 17.6.)

[16] Ezen pragmák implementálása azonban nem kötelező a fordítóprogramok számára.

Felhasználási területek

A védett egységek különös figyelmet kapnak néhány alkalmazási területen. Ezeket az alkalmazási területeket [1] az Annex-ekben írja le. A valós idejű rendszerek a D, az elosztott rendszerek az E jelű Annex-ekben kerültek bemutatásra. Mi most a C.3-at emeljük ki, a megszakítások kezelésével foglalkozó részt.

A hardver és az operációs rendszer által kiváltott megszakítások lekezelésére az Ada a védett műveleteket használhatja. Egy megszakításhoz hozzárendelhetjük egy védett objektum egy műveletét. Amikor a megszakítás kiváltódik, a védett művelet törzse fut le válaszul.

A paraméter nélküli védett eljárások nagyon kényelmes eszközt nyújtanak megszakításkezelő rutinok írására. Az `Attach_Handler` pragma segítségével kapcsolhatjuk össze az általunk elkészített védett eljárást a megfelelő megszakítás azonosítójával:

16.11. Példa

```
protected type Megszakítás_Kezelés(
         Megszakítás: Interrupt_ID) is
  procedure Válasz;
  pragma Attach_Handler(Válasz,Megszakítás);
end Megszakítás_Kezelés;

protected body Megszakítás_Kezelés is
  procedure Válasz is
  begin
    ...
  end Válasz;
end Megszakítás_Kezelés;
```

Az `Interrupt_Handler` pragma és az `Ada.Interrupts` csomagban definiált műveletek (pl. az `Attach_Handler` eljárás segítségével az összekapcsolást dinamikusan (futási időben) is elvégezhetjük.

16.3.1. Értékelés

A védett objektumok tulajdonságait két aspektusból vizsgáljuk meg. Egyrészt az érdekel minket, hogy mennyire támogatják a biztonságos programfejlesztést, másrészt, hogy milyen hatékony programok írását teszik lehetővé.

Szinkronizációs eszközként leggyakrabban a szemaforokat és a monitorokat szokták példaként felhozni. A *szemaforok*, noha működésük könnyen megérthető, nagy odafigyelést igényelnek a programozás során. Mivel az összetartozó `P` és `V` műveletek meghívása különböző helyen történik a kódban, a programozó könnyen elfeledkezhet valamelyikről: a szemaforok túlságosan alacsony szintű eszközök.

A Hoare által 1973-ban javasolt *feltételes kritikus szakasz*, és még inkább a *monitor* sokkal magasabb szintű, biztonságosabban használható szinkronizációs szerkezet. Azonban a feltételes kritikus szakasz nem támogatja az adatok és a rajtuk értelmezett műveletek összefogását, egységbezárását, és a monitor használata is okozhat gondot: a szinkronizációs feltételekhez rendelt `Vár` és `Jelez` műveletek megfelelő helyen történő aktiválása könnyen elrontható.

A védett objektumok egyesítik az említett jó tulajdonságokat a rossz tulajdonságok kiküszöbölése mellett. Az adatok és műveleteik egységbezárását a monitoroktól, az őrfeltételek használatát a feltételes kritikus szakasztól örökölte. A védett objektumok használatával, az őrfeltételeknek köszönhetően programjaink helyességének biztosítása és bizonyítása sokkal könnyebbé válik.

A védett műveleteket végrehajtó taszkok

A védett objektumok hatékony nyelvi implementálása viszonylag könnyű. Vezérlésükhöz nincs szükség külön taszkokra: a védett objektumok passzív szerkezetek. Meg lehet mutatni, hogy egy időosztásos rendszeren a védett objektumok használata sokkal kevesebb kontextus-váltást igényel, mint ugyanannak a feladatnak taszkokkal és randevúkkal, vagy akár szemaforokkal történő megoldása. (Lásd [3])

Tegyük fel, hogy van egy védett objektumunk. A taszkok versengenek műveletei végrehajtásának jogáért. Ennek a szabályait már ismerjük: az a taszk, amelyik megkapja a megfelelő kulcsot, megkezdheti a kívánt művelet végrehajtását. A csak olvasni akaró taszkok egyszerre is dolgozhatnak, az írási joggal is rendelkezők a kölcsönös kizárás előnyeit élvezhetik. A fogós kérdés az, hogy hogyan történhet a kétszintes védelem biztosítása?

A kérdés akkor merül fel, amikor egy védett eljárás vagy belépési pont törzs véget ér. Ekkor ugyanis, mielőtt megkezdődne a versengés a kulcsért, a bejutásért, a már bent levő, a várakozási sorokban álló taszkok lehetőséget kapnak az újraindulásra. Tegyük fel, hogy van is ilyen taszk. Vegyük észre, hogy nincs arra szükség, hogy gyakorlatilag is ez a taszk hajtsa végre a kívánt műveletet: végrehajthatja helyette az éppen véget ért műveletű taszk is. Ezzel a megfelelő kontextus-váltást megtakarítottuk.

16.4. Példák

A [3] alapján két védett egység deklarációt mutatunk be. Az első a korlátos buffer egy lehetséges megvalósítása: ez egy gyakran felemlegetett példa folyamatok szinkronizációjára. A második egy üzenetszórásos szignál implementációja. Ezen keresztül illusztráljuk a `requeue` utasítás hatását (343. o.) és a kétszintű védelem mechanizmusát (16.2.6.).

16.4.1. Korlátos buffer

Olyan sor típust valósítunk meg, amely párhuzamos környezetben is használható: a betevő és kivevő műveletek szinkronizáltak. A betevő művelet csak akkor hajtódhat végre, ha a sor nincs tele (ezért korlátos a buffer), szimmetrikusan, a kivevő művelet csak akkor, ha a sor nem üres. A műveletekre ezen kívül természetesen kölcsönös kizárás is biztosított.

A buffer maximális mérete a típus diszkriminánsa. A sorban tárolt elemek száma, `Be-Ki`, mindig `0` és `Max` között van.

16.12. Példa

```
protected type Korlátos_Buffer( Max: Positive ) is
  entry Betesz( X: in  Elem );
  entry Kivesz( X: out Elem );
private
  Adatok: Vektor(1..Max);
  Be, Ki: Positive := 1;
end Korlátos_Buffer;

protected body Korlátos_Buffer is
  entry Betesz( X: in Elem ) when Be-Ki < Max is
  begin
    Adatok(Be) := X;
    Be := Be + 1;
  end Betesz;

  entry Kivesz( X: out Elem ) when Be-Ki > 0 is
  begin
    X := Adatok(Ki);
    Ki := Ki + 1;
    if Ki > Max then
      Ki := Ki - Max; Be := Be - Max;
    end if;
  end Kivesz;
end Korlátos_Buffer;
```

16.4.2. Üzenetszórásos szignál

Egy esemény bekövetkeztét folyamatok várják. Amikor az esemény megtörténik, minden rá váró folyamat értesítést kap erről.

Az `Esemény` egy védett objektum. A taszkok azzal jelzik, hogy várnak rá, hogy meghívják a `Vár` belépési pontját. Az esemény kiváltását a `Jelez` belépési pont meghívásával lehet kezdeményezni. Az esemény bekövetkezéséről a rá váró taszkok úgy szereznek tudomást, hogy a `Vár` belépési pont hívása véget ér.

16.13. Példa

```
protected Esemény is
  entry Vár;
  entry Jelez;
private
  entry Alaphelyzet;
  Megtörtént: Boolean := False;
end Esemény;

protected body Esemény is
  entry Vár when Megtörtént is
  begin null; end Vár;

  entry Jelez when True is
  begin
    Megtörtént := True;
    requeue Alaphelyzet;
  end Jelez;

  entry Alaphelyzet when Vár'Count = 0 is
  begin
    Megtörtént := False;
  end Alaphelyzet;
end Esemény;
```

A `Vár` belépési ponthoz tartozó törzs üres; az őrfeltétel szerint akkor hajtódhat végre, amikor a `Megtörtént` változó igazra vált: ez jelzi, hogy az esemény bekövetkezett. A változót a `Jelez` belépési pont törzse állítja igazra.

Miért van szükség az `Alaphelyzet` belépési pontra? Ezt a belépési pontot csak a védett objektumon belül használhatjuk, hiszen a védett egység átlátszatlan részében lett deklarálva. A `requeue` utasítás a `Jelez` belépési pontot meghívó taszkokat fogja betenni az `Alaphelyzet` várakozási sorába.

Az üzenetszórásos szignálnak ugyanis teljesítenie kell egy fontos kikötést. Azok a taszkok, amelyek a `Jelez` belépési ponthoz tartozó törzs végrehajtásának megkezdése után kezdenek el várni az eseményre, nem kaphatják meg az esemény bekövetkezését jelző üzenetet. Ezt a *kétszintes védelem* biztosítja. Amikor végrehajtódhat a `Jelez` törzse, a `Megtörtént` változó értéke igaz lesz. A `Jelez`-t végrehajtó taszk bekerül az `Alaphelyzet` várakozási sorába és felfüggesztődik.

Mielőtt lemondana a – kölcsönös kizárást biztosító – író/olvasó kulcsról, megvizsgálja, hogy nem adhatná-e azt át egy várakozási sorban álló taszknak. Kiértékeli hát a többi belépési pont őrfeltételét. Ha a `Vár`-nál nem áll egy taszk sem, a `Vár'Count` nulla, tehát az `Alaphelyzet`-nél váró, a szignált

kiváltó taszk kaphatja meg a továbbhaladási jogot. Ez a `Megtörtént` változót hamisra állítja, mielőtt véget érne, és megadná a lehetőséget a kulcsra váró, a kölcsönös kizárás miatt még kinnrekedt taszkoknak. Ha valamelyik közülük a `Vár` belépési pontot hívná, az őrfeltételt már hamisnak fogja találni.

Ha azonban a `Jelez` végrehajtásakor nem volt üres a `Vár` várakozási sora, a legelső várakozó végrehajthatja a `Vár` törzsét. Miután véget ér, átadja a jogot a következőnek, majd az megint a következőnek, és ez így megy addig, amíg minden, a `Vár`-nál álló taszk meg nem tudja, hogy a várt esemény bekövetkezett. Közben a kinti, kulcsra váró taszkoknak esélyük sem volt megkezdeni a várt műveletet. Miután az összes várakozó elfogyott, a `Vár'Count` megint csak nulla lesz, és innen már tudjuk, mi fog történni.

A kettős védelemnek köszönhetően azok a taszkok, amelyek a `Jelez` megkezdése pillanatában még nem voltak a `Vár` várakozási sorában, hiába szerették volna végrehajtani a `Vár` műveletet, nem tudták azt megtenni, csak akkor, amikor a `Megtörtént` értéke már megint hamis volt.

16.5. Feladatok

16.1. Írjon perzisztens szignálként működő védett objektumot! A perzisztens szignál működési elve a következő. Taszkok egy esemény bekövetkezésére várnak. Amikor az esemény bekövetkezik, az a taszk, amelyik az esemény bekövetkeztét észlelte, „jelez": meghívja a védett objektum `Jelez` műveletét. Ezután az eseményre váró taszkok közül egy elindulhat. Ha egy taszk sem vár az eseményre annak bekövetkeztekor, a jelzés megmarad (ezért perzisztens); az első olyan taszk, amelyik a jövőben elkezd várni az eseményre, azonnal megkapja a jogot a továbbhaladásra. Az esemény többszöri bekövetkezése olyankor, amikor egy taszk sem vár rá, egyenértékű az egyszeri bekövetkezéssel.

16.2. Szimulálja egy – kellően bonyolult, tehát védett függvényt, eljárást és belépési pontot is tartalmazó – védett objektum működését taszkokkal! Fordítson külön gondot arra, hogy a kölcsönös kizárás biztosításához alkalmazott lakat kulcsának megszerzése véletlenszerűen történjen a „pályázó" taszkok között. (Használjon ágenseket a sorbanállások megszervezésére!)

16.3. Valósítsa meg a `Kapcsolótábla` védett típust! Egy kapcsolótábla valahány – korlátos számú – ún. közvetlen kapcsolatot tárol. A közvetlen kapcsolat egy rendezett páros (mint egy él egy irányított gráfban). A típusnak három művelete van.

1. Ha még nincs tele a kapcsolótábla, fel lehet venni egy új közvetlen kapcsolatot.

2. El lehet távolítani egy közvetlen kapcsolatot.
3. Le lehet kérdezni, hogy két pont – közvetlen vagy közvetett – kapcsolatban áll-e egymással (azaz vezet-e irányított út az egyik pontból a másikba).

A tárolható közvetlen kapcsolatok számának felső korlátja legyen a típus diszkriminánsa!

16.4. Vizsgálja meg, hogy az Ön által használt fordító milyen megszakítások használatára ad lehetőséget. (Az adott implementáció által támogatott megszakítások nevei az `Ada.Interrupts.Names` csomagban vannak definiálva.) Írjon olyan programot, ami egy megszakítást lekezel! (Például UNIX rendszereken a `SIGTERM` szignál elnyelésével elérhetjük azt, hogy a `kill <pid>` parancs hatására ne álljon le a programunk.)

17. Fejezet

Reprezentációs specifikációk

Noha az Ada kialakításánál az elsődleges szempont a biztonságosság volt, gyakorlati programozási feladatoknál nem hagyható figyelmen kívül a hatékonyság kérdése sem. A biztonságosság és a hatékonyság gyakran kerülnek egymással ellentétbe, ezért egy jól használható programozási nyelvben szükség van olyan eszközökre, amelyek segítségével a programozó explicit meghatározhatja, mikor melyik szempontokat részesíti előnyben. Mindezeken túl, bármennyire jól optimalizál is egy fordítóprogram, a programozónak olyan plusz ismeretei lehetnek a feladat természetéről, amelyek birtokában sokkal jobban el tudja dönteni, hogy például egy adattípust hogyan a leghatékonyabb reprezentálni. Az ilyen többletinformációk és hatékonysági elvárások megadásának eszközeit fogjuk ebben a fejezetben áttekinteni.

17.1. Reprezentációs elemek

Egy absztrakt adattípus fizikai reprezentációjának megadására használhatóak az úgynevezett *reprezentációs elemek* . Három típusa van: reprezentációs záradék, komponens záradék és reprezentációs pragma . Ezek segítségével specifikálhatjuk, hogy a típusok és más nyelvi elemek hogyan képeződjenek le a konkrét gépi architektúrára. Megadhatunk általunk hatékonyabbnak gondolt reprezentációkat, illetve lehetőséget nyújtanak a hardver jobb kihasználására, speciális elemeinek elérésére.

Egy reprezentációs elem hatásköre vagy egy, az adott deklarációs részben deklarált és a reprezentációs elemben megnevezett entitás, vagy a fordítás menetében őt közvetlenül megelőzőleg deklarált könyvtári egység.

A reprezentációs elemek használatára vonatkozó szabályok:

- Ha egy reprezentációs elem közvetlenül szabályozza valamely tulajdonságát egy típusnak vagy altípusnak, akkor az adott reprezentációs elemet a típus teljes definíciója után kell elhelyezni ám még az előtt, hogy a típust befagyasztanánk (lásd 17.1.5.).

- Amennyiben egy entitás valamely tulajdonsága egy reprezentációs elem közvetlen szabályozása alá esik, úgy tilos bármely más, az adott tulajdonságot meghatározó reprezentációs elem használata az adott entitással kapcsolatban.
- Egy reprezentációs elemnek, amely meghatározza egy típus egy objektumának tárolására felhasználható memória méretét, elegendő helyet kell biztosítania az altípus bármely elemének elhelyezéséhez.
- Egy származtatott típus örökli a szülő típusnak minden olyan tulajdonságát, amelyet a leszármazott deklarációja előtt explicit meghatároztunk valamely reprezentációs elem segítségével, illetve azokat is, amelyeket a szülő maga is örökölt egy nagyszülőtől.

17.1.1. A `Pack` direktíva

A gyorsabb elérés érdekében a fordítóprogramok az összetett típusok komponenseit igyekeznek a memóriában szóhatárra, 4-gyel, 16-tal osztható címekre elhelyezni, így elképzelhető, hogy a típus fizikai reprezentációjában szerepelhetnek használatlan területek, úgynevezett reprezentációs rések (gap). A `Pack` direktíva arra szólítja fel a fordítót, hogy igyekezzék az összetett típusokat a lehető legkisebb helyen elhelyezni. Használata:

`pragma Pack(`< *összetett típus* >`);`

Amennyiben a direktívát egy kiterjesztett típusra alkalmazzuk, a tömör ábrázolás csak a kiterjesztett részre fog vonatkozni, a szülőtől örökölt részek ábrázolása nem változik, hogy a típusok kompatibilitása megmaradjon.

17.1.2. Reprezentációs attribútumok

Bizonyos implementációtól függő tulajdonságok értékét az úgynevezett reprezentációs attribútumok lekérdezésével tudhatjuk meg. Némelynek értékét azonban meg is határozhatjuk, attribútumdefiníciós záradékok segítségével. Formája:

< *attribútumdefiníciós záradék* >::=
 `for` < *identitás azonosító* >'< *attribútum azonosító* >
 `use` < *kifejezés* >;
 | `for` < *identitás azonosító* >'< *attribútum azonosító* > `use` < *név* >;

Meg kell jegyeznünk, hogy ilyen módon új attribútumok nem hozhatóak létre, csak olyan attribútum azonosító használható attribútumdefiníciós záradékban, amelyet az Ada Nemzetközi Szabvány explicit engedélyez. Ezek a következők:

`X` jelöljön egy objektumot, programegységet vagy címkét. Ekkor:

`X'Address` Ez az attribútum az `X` számára lefoglalt tárolóelemek közül az elsőnek a címét tartalmazza. Programegységek esetén ez a programegység törzséhez tartozó gépi kódú programra vonatkozik, címkék esetében a címkéhez tartozó utasítás kódjára. Az attribútum értékének típusa `System.Address`.

`X` legyen egy objektum vagy típus.

`X'Alignment` Az X objektum, illetve típus esetén az X típusú objektumok elhelyezésekor a fordító olyan címet választ, amely az `Alignment` értékkel osztható. Rekordkomponensek esetén a rekordon belüli offszet lesz osztható a megadott értékkel. Mindez alól kivétel, ha az objektum elhelyezése nem esik az adott implementáció hatáskörébe, azaz az objektum címét egy felhasználó által definiált allokátorral határoztuk meg, explicit adtuk meg, ellenőrizetlen típuskonverzióból származik vagy importált objektumról van szó. A fordító ebben az esetben is feltételezheti, hogy az adott cím az `Alignment` érték többszöröse, de ennél szigorúbb feltevésekkel nem élhet.

Az attribútum értéke nemnegatív, `universal_integer` típusú. A 0 érték azt jelenti, hogy az objektumok nem feltétlenül illeszkednek a tárolási egységek határaira[1].

Amennyiben az `Alignment` specifikálva van egy összetett típusra vagy objektumra, értékének oszthatónak kell lennie az összetevő típusok `Alignment` értékeivel. Ez alól kivételt jelent, hogy a `Component_Size` záradék, komponens záradék, illetve `Pack` pragma felülbírálhatja a megadott `Alignment` értéket.

`X` jelöljön egy objektumot.

`X'Size` Az `X` objektum reprezentációjának méretét adja meg bitekben. Típusa `universal_integer`. Az attribútumdefiníciós záradékban az értékét meghatározó kifejezés statikus és nemnegatív kell, hogy legyen.

Ha `S` egy típus, akkor

`S'Size` Amennyiben az `S` típus definiált, `S'Size` jelöli azt a bitekben mért méretet, amelyet az adott implementáció fog választani a következő `S` típusú objektumok számára:

- Egy `S` típusú rekordkomponens, amennyiben a rekord tömörített ábrázolású.

[1] Például logikai értékeknél nem feltétlenül szükséges, hogy byte-határon kezdődjenek, hiszen ábrázolásukhoz egyetlen bit is elegendő.

- Az `S` típusról történő ellenőrzés nélküli konverzió esetén az `S` típusú formális paraméter.

Nemdefiniált `S` típus esetén az attribútum jelentése implementációfüggő. Minden implementációban a `Boolean'Size` értéknek 1-nek kell lennie.

Legyen `X` egy tömbtípus vagy tömbobjektum (esetleg bizonyos implicit elvégzett dereferenciák után), ekkor

`X'Component_Size` jelöli `X` komponenseinek bitben mért méretét. Az attribútum értéke `universal_integer` típusú.

Egy `T` jelölt típus minden `S` altípusára definiált az

`S'External_Tag` attribútum, amely az `S'Tag` attribútum egy sztringreprezentációját adja. Az attribútum értéke implementációfüggő, ám attribútumdefiniciós záradékban megváltoztatható.

Amennyiben `T` egy taszkobjektum (esetleg bizonyos implicit elvégzett dereferenciák után), akkor

`T'Storage_Size` jelöli a taszk számára lefoglalt tárolási egységek számát. Az attribútum típusa `universal_integer`. Az attribútum értéke tartalmazza a taszkhoz tartozó esetleges stack méretét, de a nyelv nem köti ki, hogy más, a taszkhoz kapcsolódó memóriaterületek is beleszámítanak-e az értékbe. A `Storage_Size` direktívát használva megadhatunk az attribútumhoz alsó korlátot. A `pragma` csak taszkdefiníció belsejében használható. Formája:

`pragma Storage_Size(`< *kifejezés* >`);`

Megjegyzések

1. A különféle entitások alábbi attribútumai – feltéve, hogy az adott esetben értelmezve vannak –, megváltoztathatók:

`Address,`	`Size,`	`Component_Size,`
`Alignment,`	`External_Tag,`	`Small,`
`Bit_Order,`	`Storage_Pool,`	`Storage_Size,`
`Write,`	`Output,`	`Read,`
`Input,`	`Machine_Radix`	

2. Általánosságban igaz, hogy ha egy attribútum értékét megváltoztatjuk, akkor az adott attribútumra történő hivatkozás az általunk megadott értéket adja vissza. Ez alól egyetlen kivétel a `Storage_Size` attribútum, mivel annak csak alsó korlátját szabályozhatjuk. Az eltérés

ott is látható, hogy míg az összes többi attribútum értéke attributum-definíciós záradék formájában bírálható felül, addig a `Storage_Size` korlátját reprezentációs pragmával adhatjuk meg.

17.1. Példa *(Reprezentációs záradékok)*

```
Bájt : constant := 8;
Lap  : constant := 2**12;

type Közepes is range 0..65_000;
for Közepes'Size use 2*Byte;
for Közepes'Alignment use 2;

Eszköz_Azonsító : Közepes;
for Eszköz_Azonosító'Size use Közepes'Size;
for Eszköz_Azonosító'Address use
    System.Storage_Elements.To_Address(16#FFFF_02);

type Rövid is delta 0.01 range -100.0 .. 100.0;
for Rövid'Size use 15;

type Autó;
type Autó_Név is access all Car;
type Autó is
   record
      Sorszám     : Integer;
      Tulajdonos : Személy_Név;
   end record;

for Autó_Név'Storage_Size use
    2000*((Autó'Size/System.Storage_Unit) +1);

function Saját_Olvasó(Bemenet : access
                  Ada.Streams.Root_Stream_Type'Class)
         return T;
for T'Read use Saját_Olvasó;
```

17.1.3. Felsorolási típus reprezentációs záradékai

A felsorolási típus reprezentációs záradékában lehetőségünk nyílik meghatározni, hogy a felsorolási típus elemeit milyen belső kódokra képezze le a fordító. Használata a következő módon történhet:

< *felsorolási típus reprezentációs záradék* >::=
 `for` < *felsorolási típus* > `use` < *felsorolási aggregátum* >;

A felsorolási aggregátum egy egydimenziós tömbaggregátum formájában adható meg, ahol az indexek a felsorolási típus elemei, a hozzájuk rendelt értékek pedig a nekik szánt reprezentáló kódok. Ezen kódok páronként különböző értékű, fordítási időben kiértékelhető kifejezésekkel adhatók meg, típusuk valamilyen egész típus. Mindezeken felül a megadott értékeknek ki kell elégíteni a típus definíciójában implicit megadott rendezési relációt.

Megjegyzés

A belső kódok lekérdezése ellenőrzés nélküli konverzióval történhet. A típus attribútumait, mint például a `Succ`, `Pred` vagy `Pos`, a reprezentációs záradék nem befolyásolja. A `Pos` attribútum például mindig az adott típusérték sorszámát adja vissza nem pedig belső kódját.

17.2. Példa *(Felsorolási típus reprezentációjának megadása)*

```
type Mix_Code is (ADD, SUB, MUL, LDA, STA, STZ);
for Mix_Code use (ADD => 1, SUB => 2, MUL => 3,
                  LDA => 8, STA => 24, STZ => 33);
```

17.1.4. Rekordok kialakításának szabályozása

Rekord típusok kialakításakor az Ada lehetőséget ad mind a rekordon belül a komponensek, mind pedig rekordok a memóriabeli elhelyezésének szabályozására. Ám mielőtt ezekre rátérnénk, tekintsük át, hogy a rekordokkal kapcsolatban milyen további reprezentációs attribútumok állnak rendelkezésünkre. Legyen `C` egy komponense valamely `R`, nem tömbtípusú, összetett objektumnak. Ekkor definiáltak a következő attribútumok:

`R.C'Position` értéke megegyezik az `R.C'Address-R'Adress` értékkel, azaz megadja, hogy a `C` komponens az `R` összetett objektum milyen offszetű tárolási elemén kezdődik.

`R.C'First_Bit` jelöli az offszetet a `C` által érintett első tárolási elem első bitjétől, a `C` által elfoglalt első bitig.

`R.C'Last_Bit` pedig a `C` által érintett első tárolási elem első bitjétől, a `C` által elfoglalt utolsó bitig mért offszetet jelöli.

Mindhárom érték típusa `universal_integer`. Ezen három attribútumon kívül, minden `S` rekordtípushoz definiált a

`S'Bit_Order` attribútum az `S`-en belüli bitsorrendet jelöli. Az attribútum `System.Bit_Order.Bit_Order` típusú. A típusnak két értéke van: `High_Order_First`, amely esetben a tárolási elemek első bitje a legmagasabb helyiértékű, illetve `Low_Order_First` amikor a legnagyobb

indexű bit a legszignifikánsabb. Az attribútum értéke attribútumdefiníciós záradékban megváltoztatható. Alapértelmezett értéke implementációfüggő.

A rekordok kialakításával és elhelyezésével kapcsolatos kívánságainkat úgynevezett rekord reprezentációs záradék formájában írhatjuk le. Ennek helyes szintaxisa a következő:

```
< rekord reprezentációs záradék >::=
    for < rekord típus > use
    record [< mod záradék >]
    {< komponens záradék >}
    end record;
```

ahol

```
< komponens záradék >::=
  < komponensnév > at < pozíció >
                   range < első bit > ... < utolsó bit >
```

Az így megadott záradékban, a mod záradékot leszámítva, a rekord kialakítását szabályozzuk. A komponensek tárolási helyre vonatkozó attribútumai (`Position`, `First_Bit`, `Last_Bit`) a megfelelő komponenshez tartozó záradékban megadott pozíció, első bit és utolsó bit paraméterek normalizált értékei lesznek. A normalizálás során úgy változnak az értékek, hogy az első bit értéke kisebb lesz mint egy tárolási egység mérete.

Rekord kiterjesztés esetén a kiterjesztésben megadott reprezentációs záradék nem definiálhatja felül az örökölt rész kialakítását. Amennyiben az ősben megadtunk valamilyen kialakítást, azt örökli a leszármazott.

Egy rekord minden komponenséhez legfeljebb egy komponens záradék tartozhat, beleértve ebbe az esetleges diszkriminánsokat is. Az így kiosztott tárterületek nem fedhetik át egymást kivéve ha olyan komponensekhez tartoznak, amelyek egy variánsrekord különböző variánsaihoz tartoznak.

Megjegyzés

Amennyiben egy komponenshez nem adtunk meg záradékot, úgy elhelyezéséről az adott implementáció szabadon rendelkezhet.

17.3. Példa *(Rekordtípus reprezentációs záradékai)*

```
   Szó  : constant := 4;  --  a tárolási elemek byte-ok,
                          --  4 byte-os szavakkal dolgozunk

   type Állapot        is (A,M,W,P);
```

```
type Üzemmód       is (Fix, Dec, Exp, Signif);

type Bájt_Maszk    is array (0..7)    of Boolean;
type Állapot_Maszk is array (Állapot) of Boolean;
type Üzemmód_Maszk is array (Üzemmód) of Boolean;

type Program_Státusz_Szó is
   record
      Rendszer_Maszk      : Bájt_Maszk;
      Védelmi_Kulcs       : Integer range 0 .. 3;
      Gépi_Állapot        : Állapot_Maszk;
      Megszakítás         : Megszakítási_Kód;
      Ilc                 : Integer range 0 .. 3;
      Cc                  : Integer range 0 .. 3;
      Program_Maszk       : Üzemmód_Maszk;
      Példány_Cím         : Mutató;
   end record;

for Program_Státusz_Szó use
   record
      Rendszer_Maszk   at 0*Word range 0  .. 7;
                                -- 8,9. bitek nem használtak
      Védelmi_Kulcs    at 0*Word range 10 .. 11;
      Gépi_Állapot     at 0*Word range 12 .. 15;
      Megszakítás      at 0*Word range 16 .. 31;
                                -- a második gépi szó
      Ilc              at 1*Word range 0  .. 1;
      Cc               at 1*Word range 2  .. 3;
      Program_Maszk    at 1*Word range 4  .. 7;
      Példány_Cím      at 1*Word range 8  .. 31;
   end record;

for Program_Státusz_Szó'Size use 8*System.Storage_Unit;
for Program_Státusz_Szó'Alignment use 8;
```

17.1.5. Fagyasztási szabályok

A fagyasztási szabályok olyan pontokat definiálnak a program szövegében ahol minden deklarált entitás „fagyasztottá" válik. Egy entitás fagyasztása azt jelenti, hogy attól a ponttól fogva az entitás fizikai reprezentációja egyértelműen meghatározott, megváltoztatására nincs mód. Általában egy entitás fagyasztását az adott entitás valamilyen felhasználása váltja ki. Néhány jellegzetes ok, ami valamely entitás fagyasztását kiválthatja:

- Ilyen lehet a nevével történő hivatkozás, vagy típus esetén egy, ahhoz a típushoz tartozó, kifejezés.
- Egy deklarációs rész, vagy védett törzs vége fagyasztja az összes bennük deklarált entitást, kivéve a hiányos típusdefiníciókat.
- Egy sablon példányosítása fagyasztja az aktuális paramétereket.
- Rekordtípus kiterjesztése fagyasztja az őstípust.
- Allokátor hívása fagyasztja a referált típust. Amennyiben az allokátor származtatott típushoz tartozik, úgy az összes őstípus is fagyasztódik.

17.2. A System csomag

Minden implementációban van egy standard könyvtári csomag, melynek neve `System`. Ez a csomag tartalmazza az adott implementáció platformfüggő definícióit.

```
package System is
   pragma Preelaborate(System);

   type Name is implementációfüggő-felsorolási-típus;
   System_Name : constant Name := implementációfüggő;


   -- Rendszerfüggő nevesített számkonstansok:

   Min_Int               : constant := root_integer'First;
   Max_Int               : constant := root_integer'Last;

   Max_Binary_Modulus    : constant := implementációfüggő;
   Max_Nonbinary_Modulus : constant := implementációfüggő;

   Max_Base_Digits       : constant := root_real'Digits;
   Max_Digits            : constant := implementációfüggő;

   Max_Mantissa          : constant := implementációfüggő;
   Fine_Delta            : constant := implementációfüggő;

   Tick                  : constant := implementációfüggő;


   -- Tárolófüggő deklarációk:
```

```
   type Address is implementációfüggő;
   Null_Address : constant Address;

   Storage_Unit : constant := implementációfüggő;
   Word_Size    : constant := implementációfüggő *
                                               Storage_Unit;
   Memory_Size  : constant := implementációfüggő;


   -- Címösszehasonlító operátorok:
   function "<" (Left, Right : Address) return Boolean;
   function "<="(Left, Right : Address) return Boolean;
   function ">" (Left, Right : Address) return Boolean;
   function ">="(Left, Right : Address) return Boolean;
   function "=" (Left, Right : Address) return Boolean;
-- function "/=" (Left, Right : Address) return Boolean;
   -- "/=" implicit definiált
   pragma Convention(Intrinsic, "<");
   ... -- és így tovább az összes, a nyelv által definiált
       -- alprogramra ebben a csomagban.

   -- Más, a nyelv által definiált deklarációk:
   type Bit_Order is (High_Order_First, Low_Order_First);
   Default_Bit_Order : constant Bit_Order;

   -- Prioritáskezeléssel kapcsolatos deklarációk:
   subtype Any_Priority is Integer range
                               implementációfüggő;
   subtype Priority is Any_Priority range
           Any_Priority'First .. implementációfüggő;
   subtype Interrupt_Priority is Any_Priority range
           Priority'Last+1 .. Any_Priority'Last;
   Default_Priority : constant Priority := (Priority'First
                                          + Priority'Last)/2;

private
   ... -- implementációfüggő
end System;
```

A `Name` típus felsorolási típus. Értékei az adott implementáció által támogatott hardverkonfigurációk. A `System_Name` konstans tartalmazza az aktuális konfigurációt.

A nevesített konstans számértékek `universal_integer` típusúak, kivéve a `Fine_Delta` és a `Tick` értékeket, amelyek a `universal_real` típushoz tartoznak.

A konstansok jelentése:

`Min_Int` A legkisebb megengedett előjeles egész érték.

`Max_Int` A legnagyobb megengedett előjeles egész érték.

`Max_Binary_Modulus` Értéke kettőhatvány. Ő és a nála kisebb pozitív kitevőjű kettőhatványok szerepelhetnek modulusként valamely moduláris típusdefinícióban.

`Max_Nonbinary_Modulus` Legfeljebb ekkora pozitív egész szerepelhet modulusként moduláris típusdefinícióban.

`Max_Base_Digits` Lebegőpontos típus definíciójában megadható maximális decimális pontosság.

`Max_Digits` Lebegőpontos típus definíciójában megadható maximális decimális pontosság, amennyiben az értéktartomány nincs korlátozva. `Max_Digits` értéke mindig kisebb vagy egyenlő `Max_Base_Digits` értékénél.

`Max_Mantissa` Egy felhasználó által definiált fixpontos típus mantisszájának maximális mérete bitekben.

`Fine_Delta` A -1.0 .. 1.0 intervallumon definiálható legnagyobb fixpontos pontosság `delta` értéke.

`Tick` Az a becsült időtartam amely alatt a `Calendar.Clock` értéke nem változik. Az érték másodpercben van megadva.

`Storage_Unit` Egy tárolási elem bitekben mért mérete.

`Word_Size` Egy gépi szó bitekben mért mérete.

`Memory_Size` Az adott gépi konfiguráció memóriájának mérete tárolási elemekben mérve.

17.3. Assembly részek beillesztése

Az Ada lehetőséget kínál arra, hogy a programjainkba Assembly nyelvű kódrészleteket illesszünk. Az ilyen részletek csak alprogramok törzsében jelenhetnek meg. Amennyiben valamely alprogram törzsében szerepel kódutasítás, úgy az alprogram törzse csak ilyen utasításokat tartalmazhat, deklarációs részében csak `use` záradékok szerepelhetnek és nem adható hozzá

kivételkezelő rész. A megjegyzések és pragmák a szokásos módon használhatóak.

A kódutasítás úgynevezett minősített kifejezés formájában adható meg. A minősített kifejezésnek valamely, a System.Machine_Code csomagban deklarált típushoz kell tartoznia. Éppen ezért kódutasítás csak olyan with záradék hatáskörében használható, amelyben a System.Machine_Code csomag szerepel.

A System.Machine_Code csomag tartalma, akár csak a kódutasítások jelentése, implementációfüggő. Általában egy minősített kifejezés egy gépi utasítást vagy assembly direktívát jelöl.

Megjegyzés

A gépi kódú függvények kivételt jelentenek azon szabály alól, hogy minden függvényben szerepelnie kell return utasításnak. Sőt, egyáltalán nem szerepelhet bennük, mivel az ilyen alprogramok törzsében csak kódutasítások használhatóak.

17.4. Példa *(Gépi kódú részek beillesztése)*

```
M : Maszk;
procedure Maszk_Beállítás;
pragma Inline(Maszk_Beállítás);

procedure Maszk_Beállítás is
        -- feltéve hogy a "with System.Machine_Code;"
        -- sor szerepel valahol feljebb:
   use System.Machine_Code;
begin
   SI_Format'(Code => SSM, B => M'Base_Reg, D => M'Disp);
   --  Base_Reg és Disp implementációfüggő attribútumok.
end Maszk_Beállítás;
```

17.4. Ellenőrzés nélküli adatelérés

Noha az Ada erősen típusos nyelv, amely meglehetősen szigorúan ellenőrzi az adathozzáférést, mégis lehetőséget ad ellenőrzés nélküli adatmanipulációkra.

17.4.1. Ellenőrzés nélküli típuskonverzió

Az Unchecked_Conversion függvénysablon megfelelő példányával ellenőrzés nélküli típuskonverziót végezhetünk két adattípus között.

```
generic
   type Source(<>) is limited private;
   type Target(<>) is limited private;
function Ada.Unchecked_Conversion(S: Source) return Target;
pragma Convention(Intrinsic, Ada.Unchecked_Conversion);
pragma Pure(Ada.Unchecked_Conversion);
```

Az `S` formális paraméter mérete megegyezik altípusának `Size` attribútumával. Ez az altípus megegyezik a `Source`-ban átadott aktuális típussal, kivéve ha az valamely megszorítás nélküli összetett típus (határozatlan indexhatárú tömb vagy diszkriminánsos rekord). Ilyen esetekben az altípus az `S` aktuális indexhatáraival vagy diszkriminánsával megszorított altípussal egyezik meg.

Mindezek figyelembe vételével, ha az alábbiak teljesülnek, akkor az ellenőrzés nélküli típuskonverzió visszatérési értéke egy olyan, a `Target` típushoz tartozó objektum, amelynek reprezentációja megegyezik az `S` forrásobjektum reprezentációjával. A sikeres konverzió feltételei:

- `S'Size = Target'Size`
- `S'Alignment = Target'Alignment`
- A `Target` típus nem valamilyen korlátozás nélküli típus.
- Mind az `S` típusának mind a `Target` típusnak összefüggő (folyamatos) a fizikai reprezentációja.
- Az `S` reprezentációja megfelel valamely `Target` típusú objektum reprezentációjának.

Ha e feltételek valamelyike sérül, úgy a hatás nem definiált. Az így elvégzett konverzió könnyen eredményezhet inkonzisztens objektumokat, azaz olyanokat, amelyek egyetlen, a típushoz tartozó, absztrakt objektumnak sem felelnek meg.

Inkonzisztens adatokat nem csak ellenőrzés nélküli konverzió után kaphatunk, hanem inputként, más nyelven implementált függvények visszatérési értékeként stb. Általában ilyenkor a hiba nem azonnal jelentkezik, mivel az adat inkonzisztenciája csak egy későbbi hivatkozásnál okoz zavart. Az ilyen futási idejű hibák kiszűrésére használható a `Valid` attribútum. Legyen `X` egy skalár objektum (esetleg bizonyos implicit dereferenciák után). Ekkor definiált az

`X'Valid` attribútum, amely igaz (`True`) érétket ad vissza, ha az `X` által jelölt objektum normális, reprezentációja érvényes.

Érvénytelen adat a következő módokon keletkezhet:

- Inicializálatlan skalár objektum.
- Ellenőrzés nélküli típuskonverzió.
- Hibás input.
- Más nyelven írt alprogram visszatérési értéke.
- Értékadás elabortálása.
- Értékadás érvénytelenítése valamely, a nyelv által definiált ellenőrzés során felfedezett hiba miatt. Ilyenkor ugyanis nem mindig váltódik ki valamilyen kivétel, csupán a célváltozó értéke lesz definiálatlan.
- Konkrét fizikai címen elhelyezett objektumra történő hivatkozás esetén.

Az `X'Valid` hivatkozás nem számít az `X` objektum olvasásának, így nem történik előzetes ellenőrzés az adat érvényességére vonatkozóan, ezért hibás objektum esetén sem lép fel futási idejű hiba.

17.4.2. Nem ellenőrzött referenciaképzés

Az Adában a dinamikus memóriakezelés a mutató (access) típusokkal oldható meg. A mutató típus sokkal biztonságosabb, mint a hagyományos pointer típusok, ám a biztonságosságnak ára van. Igen szigorú szabály, hogy egy, a blokkstruktúrában magasabb szinten elhelyezkedő mutató típushoz tartozó változó nem kaphatja értékül egy alacsonyabb szinten elhelyezkedő objektum referenciáját, mivel annak megszűnte után a változó nem létező objektumra hivatkozna. Ugyanakkor néha szükség lehet ilyen jellegű értékadásra. A nem ellenőrzött referenciaképzés pontosan erre ad lehetőséget, a programozó kezébe helyezve a felelősséget, hogy a hibás hivatkozásokat elkerülje. Legyen `X` egy objektum, amelyre elvégezhető a referenciaképzés (`aliased` objektum). Ekkor az

`X'Unchecked_Access` attribútum mindenben megegyezik az `X'Access` attribútummal, kivéve azt, hogy az `X` mindenféle hozzáférési szabályok és ellenőrzések szempontjából úgy viselkedik mintha közvetlen a könyvtári csomagban deklarálták volna.

Az `Unchecked_Access` attribútum nem használható alprogramok referenciájának képzésére.

17.5. Memóriagazdálkodás

Az Ada igen kényelmes és biztonságos memóriakezelési mechanizmusokat kínál a programozónak. A memória lefoglalása megfelelő ellenőrzések mellett történik, felszabadításáról pedig szemétgyűjtő mechanizmus gondoskodik. Bizonyos alkalmazásoknál ugyanakkor szükségessé válhat, hogy a programozó saját kezébe kaphassa a vezérlést. Ebben a részben ennek eszközeivel ismerkedünk meg.

17.5.1. Szabadlista-kezelés

Minden objektumokra vonatkozó mutató típushoz tartozik egy memóriatartalék vagy memóriagyűjtő. Több ilyen gyűjtő is létezhet a rendszerben de egyszerre több mutató típushoz is tartozhat ugyanaz. Amikor tárolóhelyet foglalunk le valamely allokátorral, a szükséges memóriamennyiséget ebből a tartalékból különíti el a rendszer. Az `Unchecked_Deallocation` eljárás példányai az adott típushoz tartozó gyűjtőbe juttatják vissza a felszabadított memóriát.

Egy ilyen gyűjtő egy változó, amelynek típusa a `Root_Storage_Pool` típus valamely leszármazottja. Alapértelmezésben maga a rendszer rendeli hozzá az egyes mutató típusokhoz a megfelelő memóriatartalékot, ám a programozónak lehetősége van újabb gyűjtők definiálására és valamely típushoz való hozzárendelésére. Legyen `A` egy mutató típus. Ekkor

`A'Storage_Pool` jelöli az `A` típushoz rendelt gyűjtőt. Az attribútum típusa `Root_Storage_Pool'Class`.

`A'Storage_Size` attribútum megadja az `A` típus gyűjtőjének fenntartott tárolási egységek számát. A `Storage_Size(A'Storage_Pool)` függvényhívás eredményével egyezik meg. Típusa `universal_integer`.

Más gyűjtő hozzárendelése attribútumdefiníciós záradék formájában történhet, ám ha ez megtörtént, akkor a `Storage_Size` érték nem adható meg külön.

Saját gyűjtők létrehozásához az alábbi könyvtári csomag használható:

```
with Ada.Finalization;
with System.Storage_Elements;
package System.Storage_Pools is
   pragma Preelaborate(System.Storage_Pools);

   type Root_Storage_Pool is abstract
      new Ada.Finalization.Limited_Controlled with private;

   procedure Allocate(
```

```
                  Pool : in out Root_Storage_Pool;
                  Storage_Address : out Address;
                  Size_In_Storage_Elements :
                           in Storage_Elements.Storage_Count;
                  Alignment :
                           in Storage_Elements.Storage_Count)
               is abstract;

      procedure Deallocate(
                  Pool : in out Root_Storage_Pool;
                  Storage_Address : in Address;
                  Size_In_Storage_Elements :
                           in Storage_Elements.Storage_Count;
                  Alignment :
                           in Storage_Elements.Storage_Count)
               is abstract;

      function Storage_Size(Pool : Root_Storage_Pool)
         return Storage_Elements.Storage_Count is abstract;

   private
      ... -- implementációfüggő
   end System.Storage_Pools;
```

Valamely `T` mutató típus egy allokátora a `T`-hez rendelt gyűjtőből foglalja le a szükséges memóriamennyiséget. Amennyiben ez a gyűjtő programozó által definiált objektum, úgy a memóriafoglalás az `Allocate` eljárás meghívásával történik, amely paraméterként megkapja a `T`-hez rendelt gyűjtőt. A `Size_In_Storage_Element` paraméter a lefoglalandó tárolási egységek számát adja meg, nem lehet nagyobb `D'Max_Size_In_Storage_Element` értékénél, ahol `D` a `T` által referált típus. Az `Alignment` paraméter értéke `D'Alignment` lesz, `Storage_Address`-ben visszaadott érték pedig a lefoglalt memóriaszakasz címe.

Ha `Storage_Size` értéket megadtuk egy mutató típushoz, akkor az adott típushoz tartozó gyűjtő `Storage_Size` attribútuma legalább a beállított érték lesz. A gyűjtő számára fenntartott terület felszabadul, amikor az adott típust deklaráló blokkot elhagyjuk.

Megjegyzések

1. Saját gyűjtőt úgy hozhatunk létre, hogy a `Root_Storage_Pool` típust kiterjesztjük, átdefiniáljuk az `Allocate`, `Deallocate` és `Storage_Size` műveleteket, majd létrehozzuk az új típus egy objektumát. Csak ezután rendelhetjük az új gyűjtőnket valamely mutató típus(ok)hoz.

2. Amennyiben saját gyűjtő típust szeretne használni, a programozó felelőssége gondoskodni a párhuzamosságból adódó problémák kezeléséről, különös tekintettel az alábbiakra:

 - Kölcsönös kizárás szükséges, amennyiben az allokátorokat különböző taszkokban kívánjuk használni.
 - A műveletek nem lehetnek blokkolóak, amennyiben `protected` objektumokban akarjuk őket használni.
 - Amennyiben interrupt-kezelőkben is használni kívánjuk őket, a kölcsönös kizárás mechanizmusát annak megfelelően kell elkészíteni.

3. A rendszer által definiált gyűjtőkre az `Allocate` és `Deallocate` műveletek explicit – nem allokátoron vagy az `Unchecked_Deallocation` eljárás egy példányán keresztüli – meghívása nem definiált hatású.

17.5.2. Ellenőrzés nélküli tárfelszabadítás

A szemétgyűjtő mechanizmus megkerülésének eszköze az ellenőrzés nélküli tárfelszabadítás. Használata esetén a programozó felelőssége annak elkerülése, hogy a már felszabadított területre hivatkozás történjék. A következő könyvtári eljárás ad lehetőséget az ellenőrzés nélküli tárfelszabadításra:

```
generic
   type Object(<>) is limited private;
   type Name   is access  Object;
procedure Ada.Unchecked_Deallocation(X : in out Name);
pragma Convention(Intrinsic, Ada.Unchecked_Deallocation);
pragma Preelaborate(Ada.Unchecked_Deallocation);
```

Tekintsük az `Ada.Unchecked_Deallocation` alábbi példányát:

17.5. Példa *(Ellenőrzés nélküli memóriafelszabadítás)*

```
type T;
type AT is access T;

procedure Free is new Ada.Unchecked_Deallocation(T, AT);

X : AT;
```

A `Free` eljárás hatásai a következők:

- A `Free(X)` végrehajtása után `X` értéke `null`.
- Amennyiben `X` értéke eredetileg is `null` volt, úgy a `Free(X)` hívásnak nincs hatása.

- Ha `X` értéke nem `null` akkor `Free(X)` először végrehajtja az `X`-hez tartozó `Finalize` eljárást, ha van, majd utána felszabadítja a hozzá tartozó memóriaterületet. Ez alól egyetlen kivétel, ha a felszabadítandó objektum taszkot is tartalmaz, mivel ilyen esetben az objektum nem szabadítható fel. Ha az adott mutató típushoz a programozó rendelt memóriagyűjtőt, akkor a felszabadítás során meghívódik a `Deallocate` művelet.

17.5.3. A `Controlled` pragma

A szemétgyűjtő mechanizmusnak joga van azon objektumok felszabadítására, amelyekre már nem történik hivatkozás, illetve joga van áthelyezni az objektumokat. Ez időnként nem kívánatos számunkra. A `Controlled` reprezentációs pragmával egy mutató típus által referált objektumokat kivonhatjuk a szemétgyűjtő eljárás hatásköréből. Amennyiben a `Controlled` pragmát specifikáljuk egy mutató típusra, amelyhez standard memóriagyűjtő tartozik, úgy a gyűjtő objektumait a szemétgyűjtésnél figyelmen kívül hagyja a rendszer. A pragma formája:

`pragma Controlled(`< *típusnév* >`);`

17.6. További fordítási direktívák

Az előző részekben találkoztunk néhány reprezentációs pragmával, amelyek a különböző entitások fizikai reprezentációjának valamely tulajdonságát befolyásolták. Ebben a részben áttekintjük a nyelv által definiált fordítási direktívák közül a fontosabbakat. Ezek a fizikai reprezentáción túl a kódgenerálás tulajdonságait, paraméterek átadási módjait és még sok egyéb tulajdonságot befolyásolnak, melyek hatékonyabb vagy kiszámíthatóbb programműködést eredményezhetnek, vagy éppen a más nyelven írt rutinkönyvtárakkal való kapcsolatkialakítást szolgálják.

`Atomic` A direktíva valamely típus vagy objektum atomicitásának kijelölésére szolgál. Típus megadása esetén a típus minden objektuma atomi lesz. Atomi objektumok esetén mind az írási mind az olvasási műveletek megszakíthatatlan módon hajtódnak végre az adott objektumon. A direktíva használata:

`pragma Atomic(`< *entitásnév* >`);`

`Atomic_Components` Ez a direktíva egy tömb minden komponensének atomicitását jelöli ki. Használata:

`pragma Atomic_Components(`< *tömbnév* >`);`

`Convention` A direktíva segítségével arra szólíthatjuk fel a fordítót, hogy valamely más nyelv konvencióit kövesse. Például a

```
pragma Convention(Fortran, Matrix);
```

sorral arra utasíthatjuk a rendszert, hogy a Fortran konvencióknak megfelelően a `Matrix` típust oszlopfolytonosan tárolja. A használható *konvencióazonosító*k nagy része implementációfüggő, leszámítva kettőt, amely a nyelv definíciójának része. Ezek:

- `Ada` – ez az alapértelmezett konvenció.
- `Intrinsic` – olyan hívásokat jelent, amelyek be vannak építve a fordítóprogramba. Ilyen például a „/=" operátor, amely implicit definiálódik az „="-ből, azon attribútumok, amelyek maguk is alprogramok stb.

A direktíva használatának módja:

`pragma Convention(`< *konvencióazonosító* >, < *entitásnév* >);

`Elaborate` Használatával kiköthetjük, hogy a felsorolt *könyvtári egységek* törzsének kiértékelése az aktuális *könyvtári egység* kiértékelése előtt megtörténjen.

`pragma Elaborate(`< *könyvtári egység* > {, < *könyvtári egység* >});

`Elaborate_All` Használatával kiköthetjük, hogy a felsorolt *könyvtári egységekhez* szükséges összes egységek törzsének kiértékelése az aktuális *könyvtári egység* kiértékelése előtt megtörténjen.

`pragma Elaborate_All(` < *könyvtári egység* >
{, < *könyvtári egység* >});

`Elaborate_Body` Kiköti, hogy a *könyvtári egység* törzse rögtön a deklarációk után kiértékelésre kerüljön.

`pragma Elaborate_Body`[(< *könyvtári egység* >)];

`Export` A direktíva segítségével arról informálhatjuk fordítót, hogy a megnevezett entitást az Ada programon kívül, más nyelvi környezetben is használni akarjuk. A használható *konvencióazonosító*król részletesen a `Convention` pragma leírásánál olvashatunk. A *külső név* az a név, amelyen a külső rendszerben az entitásra hivatkozni szeretnénk. A *szerkesztési név* pedig az az elnevezés amelyet a megcélzott idegen

nyelv fordítóprogramja ad meg a linkernek az entitás azonosítására. A direktíva használatának módja:

`pragma Export(`< *konvencióazonosító* >, < *entitásnév* >
[, < *külső név* >][, < *szerkesztési név* >]);

`Import` Az `Export` direktíva fordítottja, segítségével idegen környezetben definiált entitásra hivatkozhatunk. A paraméterek értelmezését lásd ott.

`pragma Import(`< *konvencióazonosító* >, < *entitásnév* >
[, < *külső név* >][, < *szerkesztési név* >]);

`Inline` Ajánlás a fordítónak. A nevezett alprogramok törzse a hívások helyére makrószerűen bemásolható. Amennyiben alprogram-sablon is szerepel a paraméterlistájában, úgy annak minden példányára érvényes a direktíva. A pragmát abban a deklarációs részben kell elhelyezni, ahol a megnevezett alprogramok specifikációi szerepelnek.

`pragma Inline(`< *név* > {, < *név* >});

`Linker_Option` A direktíva valamely fordítási egység deklarációs részében helyezhető el. Hatására a *sztringkifejezés* paraméterként átadódik a linkernek akkor, amikor a pragmát közvetlenül tartalmazó fordítási egység kódját hozzáveszi az adott partícióhoz.

`pragma Linker_Option(`< *sztringkifejezés* >);

`List` Ezzel a direktívával a fordítóprogramban a listagenerálást kapcsolhatjuk ki és be. Az *azonosító* értéke `On` vagy `Off` lehet. Bárhol elhelyezhető, ahol pragma szerepelhet és hatása a következő `List` pragmáig, vagy a fordítás végéig tart.

`pragma List(`< *azonosító* >);

`Optimize` Ezzel a direktívával a fordítóprogramban optimalizáció célját állíthatjuk be. Az *azonosító* értéke `Time`, `Space` vagy `Off` lehet, melyek jelentése rendre futási időre- vagy memóriahasználatra történő optimalizálás, illetve az optimalizáció kikapcsolása. Ez a pragma bárhol elhelyezhető, ahol pragma megengedett és hatása az őt tartalmazó deklarációs rész végéig, vagy ha fordítási egység helyére írtuk akkor a fordítás végéig érvényes.

`pragma Optimize(`< *azonosító* >);

`Page` Ez direktíva bárhol elhelyezhető, ahol pragmát lehet használni. Használatával elérhetjük, hogy a listagenerálás új oldalon folytatódjék, feltéve, hogy egyáltalán generál listát a fordító.

`pragma Page;`

`Preelaborate` Előzetesen kiértékelendő *könyvtári egység* megjelölése. Az előzetesen kiértékelendő egységek kiértékelése az összes nem előzetesen kiértékelendő egység feldolgozása előtt megtörténik.

`pragma Preelaborate[(< könyvtári egység >)];`

`Restrictions` Használatával felelősséget vállalhatunk bizonyos megszorítások betartására, így egyszerűsítve a futásidejű ellenőrzőrendszert. A megadható megszorítások implementációfüggőek. A pragma konfigurációs pragma, ezért, hacsak az adott megszorítás másként nem rendelkezik, az egész partíciónak meg kell felelnie a megszorításnak.

`pragma Restrictions(< megszorítás > {, < megszorítás >});`

`Suppress` A direktívát használva lehetőségünk van kikapcsolni az *azonosító*jával megadott futásidejű ellenőrzést. Az opcionális, második paraméterben kiköthetjük, hogy az ellenőrzés elhagyása mely entitásra vonatkozzon. A pragma deklarációs rész elején, csomagspecifikáció elején, vagy konfigurációs pragmaként használható. A kikapcsolható ellenőrzések:

`Access_Check,`	`Discriminant_Check,`	`Division_Check,`
`Index_Check,`	`Length_Check,`	`Overflow_Check,`
`Range_Check,`	`Elaboration_Check,`	`Tag_Check,`
`Storage_Check,`	`Accessibility_Check,`	`All_Checks`

`pragma Suppress(< azonosító >, [, [On =>] < entitás név >]);`

A. Függelék

Előre definiált nyelvi környezet

Az alábbi fejezetben megismerkedhetünk azoknak a könyvtári egységeknek a specifikációjával, amellyel minden Ada fordítónak – és a hozzátartozó környezetnek – rendelkeznie kell. Három kiindulási könyvtári egység van: az `Ada`, az `Interfaces` és a `System`. A többi könyvtár ezek valamelyikének gyermeke, ill. leszármazottja.

A továbbiakban az `Ada` csomag leszármazottaira általában minősítés nélkül fogunk hivatkozni. A nyelv tervezése során nem volt ugyan szigorú szabály arra, hogy az egyes könyvtárak mely alapkönyvtár alá tartozzanak, a lehető legtöbb csomagot megpróbálták az `Ada` csomag alatt specifikálni. Az interfészt azonban külön kategóriába sorolták, míg a rendszerfüggő - és ezért nem hordozható – csomagok kerültek a `System` alapkönyvtár alá.

A nyelv minden implementációja biztosítja, hogy a könyvtári eljárások reentránsak, azaz egy ilyen alprogram konkurens végrehajtás esetén is a specifikációjának megfelelően fog működni, feltéve, hogy a referenciaként átadott paraméterek egymástól független objektumokra hivatkoznak. Pl. A `Text_IO.Put` művelet egyidejű meghívásai egészen addig jól fognak működni, amíg különböző fájlokra vonatkoznak. Másrészről ha ezek a konkurens hívások ugyanarra a fájlra vonatkoznak, akkor ez az osztott változók hibás használatához vezethet.

A.1. A Standard csomag

A továbbiakban először a `Standard` csomaggal foglalkozunk. Ez a csomag tartalmazza a nyelv összes előre definiált azonosítóját. A csomaghoz tartozó törzs implementációfüggő. A csomagban deklarált típusok előredefiniált műveletei a csomag definíciójában – implicit deklarációjuk folytán – csak megjegyzés formában szerepelnek.

A `Standard` csomag az alábbi deklarációkat tartalmazza:

```
package Standard is
 pragma Pure(Standard);

 type Boolean is (False, True);
```

A Boolean típushoz definiált relációs és logikai operátorok:

```
-- function "=" (Left, Right : Boolean) return Boolean;
-- function "/=" (Left, Right : Boolean) return Boolean;
-- function "<" (Left, Right : Boolean) return Boolean;
-- function "<=" (Left, Right : Boolean) return Boolean;
-- function ">" (Left, Right : Boolean) return Boolean;
-- function ">=" (Left, Right : Boolean) return Boolean;

-- function "and" (Left, Right : Boolean) return Boolean;
-- function "or"  (Left, Right : Boolean) return Boolean;
-- function "xor" (Left, Right : Boolean) return Boolean;
-- function "not" (Right : Boolean) return Boolean;
```

A relációs operátorokat a felsorolási típusokra vonatkozó rendezés határozza meg, tehát `False < True`. Az egész számok típusa is ebben a csomagban található, továbbá itt szerepel a `Natural` és `Positive` altípusok deklarációja is:

```
type Integer is range implementation-defined;

subtype Natural  is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

Az `Integer` típus előre definiált operátorai:

```
-- function "="  (Left, Right : Integer'Base)
      return Boolean;
-- function "/=" (Left, Right : Integer'Base)
      return Boolean;
-- function "<"  (Left, Right : Integer'Base)
      return Boolean;
-- function "<=" (Left, Right : Integer'Base)
      return Boolean;
-- function ">"  (Left, Right : Integer'Base)
      return Boolean;
-- function ">=" (Left, Right : Integer'Base)
      return Boolean;
```

```
-- function "+" (Right : Integer'Base)
      return Integer'Base;
-- function "-" (Right : Integer'Base)
      return Integer'Base;
-- function "abs" (Right : Integer'Base)
      return Integer'Base;

-- function "+" (Left, Right : Integer'Base)
      return Integer'Base;
-- function "-" (Left, Right : Integer'Base)
      return Integer'Base;
-- function "*" (Left, Right : Integer'Base)
      return Integer'Base;
-- function "/" (Left, Right : Integer'Base)
      return Integer'Base;
-- function "rem" (Left, Right : Integer'Base)
      return Integer'Base;
-- function "mod" (Left, Right : Integer'Base)
      return Integer'Base;

-- function "**"  (Left : Integer'Base; Right : Natural)
      return Integer'Base;
```

A `root_integer` és a többi előre definiált egész típus műveleteinek specifikációját úgy kaphatjuk meg, hogy a fenti specifikációban az `Integer` típust a megfelelőre cseréljük. A hatványozás operátor jobboldali operandusának (a kitevőnek) a típusa marad `Natural`. A lebegőpontos számok és műveleteik deklarációja is ebben a csomagban található:

```
type Float is digits implementation-defined;

-- function "=" (Left, Right : Float) return Boolean;
-- function "/="  (Left, Right : Float) return Boolean;
-- function "<" (Left, Right : Float) return Boolean;
-- function "<="  (Left, Right : Float) return Boolean;
-- function ">" (Left, Right : Float) return Boolean;
-- function ">="  (Left, Right : Float) return Boolean;

-- function "+" (Right : Float) return Float;
-- function "-" (Right : Float) return Float;
-- function "abs" (Right : Float) return Float;

-- function "+" (Left, Right : Float) return Float;
-- function "-" (Left, Right : Float) return Float;
```

```
-- function "*" (Left, Right : Float) return Float;
-- function "/" (Left, Right : Float) return Float;

-- function "**"  (Left : Float; Right : Integer'Base)
      return Float;
```

A *root_real* és a többi előre definiált lebegőpontos típus műveleteinek specifikációját úgy kaphatjuk meg, hogy a fenti specifikációban a `Float` típust a megfelelőre cseréljük. A *root_integer* és *root_real* numerikus típusokhoz további – vegyes – műveletek is tartoznak:

```
function "*" (Left : root_integer; Right : root_real)
  return root_real;

function "*" (Left : root_real;  Right : root_integer)
  return root_real;

function "/" (Left : root_real;  Right : root_integer)
  return root_real;
```

Az *universal_fixed* előredefiniált típus egyedüli multiplikatív operátorai:

```
function "*" (Left : universal_fixed;
              Right : universal_fixed)
  return universal_fixed;

function "/" (Left : universal_fixed;
              Right : universal_fixed)
  return universal_fixed;
```

A karakter típus deklarációja az `ISO 8859-1` szabvány szerinti karakterkészleten alapszik. A vezérlő karaktereknek megfelelő pozíciókon a karakter literálok helyett azok rövidített neve szerepel.

```
type Character is

   (nul, soh, stx, etx, eot, enq, ack, bel,
  bs, ht, lf, vt, ff, cr, so, si,
  dle, dc1, dc2, dc3, dc4, nak, syn, etb,
  can, em, sub, esc, fs, gs, rs, us,

  ' ',  '!',  '"',  '#', '$',  '%',  '&',  ''',
  '(',  ')',  '*',  '+', ',',  '-',  '.',  '/',
  '0',  '1',  '2',  '3', '4',  '5',  '6',  '7',
```

```
'8', '9', ':', ';', '<', '=', '>', '?',

'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', '^', '_',

'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '{', '|', '}', '~', del,

reserved_128, reserved_129, bph, nbh,
reserved_132, nel, ssa, esa,

hts, htj, vts, pld, plu, ri, ss2, ss3,
dcs, pu1, pu2, sts, cch, mw, spa, epa,

sos, reserved_153, sci, csi,
st, osc, pm, apc,

... );
```

Lévén felsorolási típus, a `Character` típus előre definiált operátorai megegyeznek a felsorolási típusokra definiált műveletekkel.

A `Wide_Character` típus az `ISO 10646 BMP` karakterkészletre épül. Az első 256 pozíción megegyezik a `Character` típus tartalmával.

```
type Wide_Character is (nul, soh ... FFFE, FFFF);
```

Itt jegyezzük meg, hogy az Ada83-ban a karakteres típusok az `ASCII` csomagban voltak definiálva. Ez a csomag az Ada95-ben a `Standard` csomagba lett beágyazva.

```
package ASCII is ... end ASCII;
```

Az előre definiált `String` és `Wide_String` típusok deklarációja is – a rájuk vonatkozó operátorokkal együtt – ebben a csomagban található. Az alábbiakban csak a `String` típus műveleteit soroljuk fel, a `Wide_String` típusoknak pontosan ugyanezek a műveletei vannak.

```
type String is array(Positive range <>) of Character;
pragma Pack(String);
```

```
  -- function "="  (Left, Right: String) return Boolean;
  -- function "/=" (Left, Right: String) return Boolean;
  -- function "<"  (Left, Right: String) return Boolean;
  -- function "<=" (Left, Right: String) return Boolean;
  -- function ">"  (Left, Right: String) return Boolean;
  -- function ">=" (Left, Right: String) return Boolean;

  -- function "&" (Left: String;  Right: String)
        return String;
  -- function "&" (Left: Character; Right: String)
        return String;
  -- function "&" (Left: String;  Right: Character)
        return String;
  -- function "&" (Left: Character; Right: Character)
        return String;

  type Wide_String is array(Positive range <>)
     of Wide_Character;
  pragma Pack(Wide_String);
```

A csomag végül deklarálja még a `Duration` fixpontos típust, és néhány kivételt. A `Duration` típusra a fixpontos típusokra általában megengedett műveletek vannak értelmezve.

```
  type Duration is delta implementation-defined
       range implementation-defined;

  Constraint_Error: exception;
  Program_Error : exception;
  Storage_Error : exception;
  Tasking_Error : exception;

 end Standard;
```

A hordozhatóság szempontjából nagyon lényeges, hogy a `Standard` csomagnak nincs átlátszatlan része.

A nyelv egyes implementációi deklarálhatnak további előre definiált skalár és lebegőpontos típusokat is.

A.2. Karakterkezelés

A továbbiakban először a karakterek kezelését vizsgáljuk meg közelebbről: az üres `Characters` csomagot, továbbá a gyerek `Characters.Handling` és `Characters.Latin_1` csomagjait.

```
package Ada.Characters is
  pragma Pure(Characters);
end Ada.Characters;
```

A `Characters.Handling` csomag karakteres adatok osztályozására és konverziójára ad műveleteket. Ezen kívül néhány `Wide_Character` típust kezelő műveletet is tartalmaz.

```
package Ada.Characters.Handling is
  pragma Preelaborate(Handling);

  function Is_Control (Item : in Character) return Boolean;
```

Igaz, ha `Item` vezérlő karakter. Vezérlő karakternek számítanak a 0..31, ill. a 127..159 intervallumokba eső pozíción levő karakterek.

```
  function Is_Graphic (Item : in Character) return Boolean;
```

Igaz, ha `Item` grafikus karakter. Grafikus – azaz megjeleníthető – karakternek számítanak a 32..126, ill. a 160..255 intervallumokba eső pozíción levő karakterek.

```
  function Is_Letter  (Item : in Character) return Boolean;
```

Igaz, ha `Item` betű. Betűnek tekintjük az 'A'..'Z' és az 'a'..'z' intervallumba eső karaktereket, valamint azokat, amelyek pozíciója az alábbi intervallumok egyikébe esik: 192..214, 216..246 és 248..255.

```
  function Is_Lower (Item : in Character) return Boolean;
```

Igaz, ha `Item` kisbetű. Kisbetűnek tekintjük az 'a'..'z' intervallumba eső karaktereket, valamint azokat, amelyek pozíciója az alábbi intervallumok egyikébe esik: 223..246 és 248..255.

```
  function Is_Upper (Item : in Character) return Boolean;
```

Igaz, ha `Item` nagybetű. Nagybetűnek tekintjük az 'A'..'Z' intervallumba eső karaktereket, valamint azokat, amelyek pozíciója az alábbi intervallumok egyikébe esik: 192..214 és 216..222.

```
  function Is_Basic (Item : in Character) return Boolean;
```

Igaz, ha `Item` alapbetű. Egy karakter alapbetű, ha az 'A'..'Z' vagy 'a'..'z' intevallumok valamelyikébe esik, vagy az alábbiak valamelyike: kis vagy nagy AE kettőshangzó, kis vagy nagy izlandi eth vagy thorn hangzó, illetve német ß.

```
function Is_Digit (Item : in Character) return Boolean;
function Is_Decimal_Digit (Item : in Character)
   return Boolean
   renames Is_Digit;
```

Igaz, ha a karakter számjegy, azaz a '0'..'9' intervallumba esik.

```
function Is_Hexadecimal_Digit (Item : in Character)
   return Boolean;
```

Igaz, ha `Item` számjegy (mint fent), vagy beleesik az 'A'..'F' vagy az 'a'..'f' intervallumok valamelyikébe.

```
function Is_Alphanumeric  (Item : in Character)
   return Boolean;
```

Igaz, ha `Item` alfanumerikus karakter, azaz ha számjegy, vagy betű.

```
function Is_Special (Item : in Character) return Boolean;
```

Igaz, ha `Item` speciális grafikus karakter. Ebbe a kategóriába a nem alfanumerikus grafikus karakterek tartoznak.

```
function To_Lower (Item : in Character) return Character;
```

Ha `Is_Upper(Item)`, akkor a megfelelő kisbetűt adja vissza, ellenkező esetben a bemenő paramétert.

```
function To_Upper (Item : in Character) return Character;
```

Ha `Is_Lower(Item)` és van hozzá tartozó nagybetű, akkor a megfelelő nagybetűt adja vissza, ellenkező esetben a bemenő paramétert. A német ß-nek és az y-diaeresisnek nincs nagybetűs változata.

```
function To_Basic (Item : in Character) return Character;
```

Ha `Item` betű, de nem alapbetű, akkor az ékezetek nélküli változatot adja vissza, ellenkező esetben a bemenő paramétert.

```
function To_Lower (Item : in String) return String;
function To_Upper (Item : in String) return String;
function To_Basic (Item : in String) return String;
```

A fenti három függvény értelemszerűen a paraméterként kapott sztring összes elemére meghívja a megfelelő karakteres függvényt, és az így kapott sztringet adja vissza. A visszaadott sztring indexelése 1-gyel kezdődik. Null sztringgel történő meghívás esetén null sztringet ad vissza.

```
subtype ISO_646 is
Character range Character'Val(0) .. Character'Val(127);

function Is_ISO_646 (Item : in Character) return Boolean;
function Is_ISO_646 (Item : in String)  return Boolean;
```

A fenti függvények közül a karakter formális paraméterű azt adja vissza, hogy a paraméter beleesik-e az `ISO_646` altípusba. A sztring paraméterű akkor igaz, ha a sztring összes eleme beleesik az `ISO_646` altípusba.

```
function To_ISO_646 (Item : in Character;
       Substitute : in ISO_646 := ' ')
return ISO_646;

function To_ISO_646 (Item : in String;
       Substitute : in ISO_646 := ' ')
return String;
```

Az első függvény az `Item` paramétert adja vissza, ha az beleesik az `ISO_646` altípusba, különben pedig a `Substitute` lesz a visszatérési érték. A második függvény ugyanezt a konverziót végzi el az `Item` sztring összes elemére. A visszaadott sztring értelmezési tartománya `1.. Item'Length` lesz.

```
function Is_Character (Item : in Wide_Character)
   return Boolean;
function Is_String  (Item : in Wide_String)
   return Boolean;
```

Ha `Wide_Character'Pos(Item)<=Character'Pos(Character'Last)`, akkor az `Is_Character` függvény `True` értékkel tér vissza. Az `Is_String` függvény pedig akkor ad igazat, ha a paraméter minden egyes elemére `Is_Character` teljesül.

```
function To_Character (Item : in Wide_Character;
       Substitute : in Character := ' ')
return Character;

function To_String  (Item : in Wide_String;
       Substitute : in Character := ' ')
return String;
```

A fenti két függvény a `To_ISO_646` függvényekhez hasonlatos. Az első függvény az `Item` paramétert adja vissza, ha `Is_Character(Item)`, különben pedig a `Substitute` lesz a visszatérési érték. A második függvény ugyanezt a konverziót végzi el az `Item` sztring összes elemére. A visszaadott sztring értelmezési tartománya `1..Item'Length` lesz.

```
   function To_Wide_Character (Item : in Character)
      return Wide_Character;
```

Azt az X Wide_Character értéket adja vissza, amelyre Character'Pos(Item) = Wide_Character'Pos(X).

```
   function To_Wide_String  (Item : in String)
      return Wide_String;

 end Ada.Characters.Handling;
```

Azzal az 1..Item'Length értelmezési tartományú Wide_Stringgel tér vissza, amelynek mindegyik eleme a To_Wide_Character függvény értéke a megfelelő Item-beli elemre.

A Characters.Latin_1 csomag definícióját nem részletezzük, csupán megjegyezzük, hogy az ISO 8859-1 karakterkészlet elemeire deklarált konstansokat tartalmazza. Az egyes Ada fordítók definiálhatnak más, a lokális karakterkészletnek megfelelő gyerek csomagokat.

A.3. Sztringkezelés

A sztringkezelő csomagok alapja a Strings csomag, amely a közös deklarációkat – konstansok, kivételek és típusok definícióját – tartalmazza:

```
 package Ada.Strings is
  pragma Pure(Strings);

  Space   : constant Character      := ' ';
  Wide_Space : constant Wide_Character := ' ';

  Length_Error, Pattern_Error,
     Index_Error, Translation_Error : exception;

  type Alignment  is (Left, Right, Center);
  type Truncation is (Left, Right, Error);
  type Membership is (Inside, Outside);
  type Direction  is (Forward, Backward);
  type Trim_End is (Left, Right, Both);
 end Ada.Strings;
```

A Strings.Map csomag karakterkészletekhez, illetve a közöttük levő leképezésekhez definiál típusokat és műveleteket.

```
package Ada.Strings.Maps is
 pragma Preelaborate(Maps);

 type Character_Set is private;

 Null_Set : constant Character_Set;
```

A `Character_Set` típus egy objektuma egy karakterekből álló halmazt reprezentál. A `Null_Set` konstans az üres halmazt reprezentálja.

```
 type Character_Range is
   record
    Low  : Character;
    High : Character;
   end record;

 type Character_Ranges is array (Positive range <>)
    of Character_Range;
```

A `Character_Range` típus egy `Obj` objektuma a karaktereknek azt a halmazát reprezentálja, amelyek az `Obj.Low .. Obj.High` intervallumba esnek. A `Character_Ranges` típus egy objektuma `Character_Range` objektumok unióját reprezentálja.

```
 function To_Set  (Ranges : in Character_Ranges)
    return Character_Set;
```

Ha `Ranges'Length = 0`, akkor értéke az üres halmaz, egyébként az a karakterhalmaz, amelyet az argumentum reprezentál.

```
 function To_Set  (Span : in Character_Range)
    return Character_Set;
```

A visszatérési érték az argumentum intervallumba eső karaktereinek halmaza.

```
 function To_Ranges (Set  : in Character_Set)
    return Character_Ranges;
```

Ha `Set = Null_Set`, akkor egy üres `Character_Ranges` tömböt ad vissza, ellenkező esetben azt a legrövidebb vektort, amely a paraméter halmaz összefüggő intervallumait tartalmazza a kezdőindex szerint növekvő sorrendben.

```
 function "=" (Left, Right : in Character_Set)
   return Boolean;
```

Az egyenlőség-vizsgálat akkor ad vissza igazat, ha mindkét oldal ugyanazt a karakterhalmazt reprezentálja.

```
function "not" (Right : in Character_Set)
   return Character_Set;
function "and" (Left, Right : in Character_Set)
   return Character_Set;
function "or"  (Left, Right : in Character_Set)
    return Character_Set;
function "xor" (Left, Right : in Character_Set)
    return Character_Set;
function "-" (Left, Right : in Character_Set)
    return Character_Set;
```

A fenti logikai operátorok a velük ekvivalens halmazműveleteknek felelnek meg, azaz `not` = komplementerképzés a `Character` típus típusértékhalmazára, `and` = metszetképzés, `or` = unióképzés, `xor` = szimmetrikus differencia, `-` = halmazok különbsége.

```
function Is_In (Element : in Character;
      Set : in Character_Set)
 return Boolean;

function Is_Subset (Elements : in Character_Set;
       Set   : in Character_Set)
 return Boolean;
function "<=" (Left  : in Character_Set;
     Right : in Character_Set)
 return Boolean renames Is_Subset;
```

Az `Is_In` művelet pontosan akkor ad igazat, ha `Element` benne van `Set`-ben. Hasonlóan az `Is_Subset` művelet pontosan akkor tér vissza igaz értékkel, ha `Elements` részhalmaza `Set`-nek. Ez utóbbi műveletre bevezetjük a `<=` infix operátort is.

A karakterhalmazok reprezentálására egy másik megközelítés a karakter tömb, a sztring. Bár a `Character_Sequence` típus következő definíciója okoz némi redundanciát, a műveletekben kihangsúlyozza a paraméter célját.

```
subtype Character_Sequence is String;
function To_Set (Sequence  : in Character_Sequence)
   return Character_Set;
function To_Set (Singleton : in Character)
   return Character_Set;
```

A Sequence objektum tulajdonképpen azt a karakterhalmazt ábrázolja, amelyet a benne levő karakterek képeznek (természetesen a többször előfordulókat csak egyszer figyelembe véve). Ugyanígy a Singleton objektum megfelel egy egyelemű halmaznak. A To_Set függvény ezeket a természetes megfelelőket adja vissza függvényértékként.

```
function To_Sequence (Set  : in Character_Set)
   return Character_Sequence;
```

A To_Sequence függvény azt – a paraméterként megadott halmaz elemeiből álló növekvően rendezett – vektort adja vissza, amely a halmaz minden elemét pontosan egyszer tartalmazza.

```
type Character_Mapping is private;

function Value (Map : in Character_Mapping;
      Element : in Character)
 return Character;

Identity : constant Character_Mapping;
```

A Character_Mapping típus egy objektuma egy Character->Character injektív leképezést reprezentál. A Value függvényen keresztül kaphatjuk meg azt, hogy a Map paraméter az Element paramétert mire képezi le. Az Identity konstans azt a leképezést reprezentálja, amely minden karaktert önmagára képez le.

```
function To_Mapping (From, To : in Character_Sequence)
   return Character_Mapping;
```

A To_Mapping függvény egy olyan leképezés objektumot állít elő, amely a From vektor minden eleméhez a To vektor megfelelő elemét, míg minden más karakterhez önmagát rendeli. Translation_Error kivételt vált ki, ha a két vektor hossza különböző, vagy egy karakter a From vektorban egynél többször szerepel.

```
function To_Domain (Map : in Character_Mapping)
   return Character_Sequence;
function To_Range  (Map : in Character_Mapping)
   return Character_Sequence;
```

A To_Domain függvény azt a legrövidebb karaktertömböt adja vissza, amely növekvő sorrendben tartalmazza az összes olyan karaktert, amelyet a paraméter leképezés nem önmagára képez le. A visszaadott érték indexelése

1-gyel kezdődik. A To_Range függvény egy pontosan olyan értelmezési tartományú vektort ad vissza, mint a To_Domain függvény, oly módon, hogy a paraméter Map leképezés a To_Domain függvény értékének minden eleméhez a To_Range függvény értékének ugyanannyiadik elemét rendeli.

```
  type Character_Mapping_Function is
    access function (From : in Character) return Character;
```

A Character_Mapping_Function típus egy F objektuma a C karaktert az F.all(C) értékre képezi le, azaz ez a függvénytípus is a Character -> Character injektív leképezéseket reprezentálja.

```
 private
    ...
 end Ada.Strings.Maps;
```

A Character_Mapping és Character_Mapping_Function típusok mindegyikét használhatjuk a kereső eljárásokban (pl. kisbetű/nagybetű különbségek elfedésére).

A.3.1. Fix hosszúságú sztringek kezelése

A következőkben a fix-hosszú sztringek kezelését tekintjük át. Fix hosszú sztringnek tekintjük a Standard.String típus értékeit. Az ezeket a sztringeket kezelő műveletek a Strings.Fixed csomagban találhatóak. Ezeknek a műveleteknek általában két formája is van: az egyik egy függvény sztring visszatérési értékkel, a másik pedig eljárás egy in out String paraméterrel. Míg a függvények általában elegánsabban használhatók, az eljárások hatékonyabbak, hiszen általában megspórolnak némi másolást és dinamikus memória allokációt. A String visszatérési értékű függvények visszaadott értékének alsó indexe mindig 1.

```
 with Ada.Strings.Maps;
 package Ada.Strings.Fixed is
  pragma Preelaborate(Fixed);

  procedure Move (Source  : in  String;
        Target  : out String;
        Drop   : in  Truncation := Error;
        Justify : in  Alignment  := Left;
        Pad : in  Character   := Space);
```

A Move eljárás karaktereket másol a Source stringből a Target sztringbe az alábbi szabályok szerint:

– ha a két sztring egyforma hosszú, akkor a `Target` stringnek értékül adja a `Source` stringet;

– ha `Source` rövidebb mint `Target`, akkor a `Justify` paraméter értékétől függően `Source` `Target` elejére (`Left`), végére (`Right`) vagy a közepére (`Center`, ekkor páratlan hossz-eltérés esetén a jobb oldalon lesz eggyel több üres hely) másolódik. A `Source`-ból ki nem töltött pozíciókat `Pad` karakterrel tölti fel.

– ha `Source` hosszabb mint `Target`, akkor a `Drop` paraméter értékétől függően a `Source` sztring elejét (Left), vagy végét (`Right`) vágja le úgy, hogy beleférjen `Target`-be. Ha `Drop = Error`, akkor a viselkedést a `Justify` és `Pad` paraméterek értéke határozza meg:

– ha `Justify` értéke `Left`, és `Source` jobb széléről csak `Pad` karakterek vágódnának le, akkor `Source` eleje másolódik át.

– ha `Justify` értéke `Right`, és `Source` bal széléről csak `Pad` karakterek vágódnának le, akkor `Source` vége másolódik át.

– minden más esetben `Length_Error` kivétel váltódik ki.

```
function Index (Source : in String;
      Pattern  : in String;
      Going  : in Direction := Forward;
      Mapping  : in Maps.Character_Mapping
           := Maps.Identity)
 return Natural;

function Index (Source : in String;
      Pattern  : in String;
      Going  : in Direction := Forward;
      Mapping  : in Maps.Character_Mapping_Function)
 return Natural;
```

Mindkét `Index` függvény a `Source` paraméterben keresi, hogy a `Mapping` leképezést figyelembe véve hol illeszkedik rá a `Pattern` paraméter. A `Going` paraméter határozza meg a keresés irányát. Ha értéke `Forward`, akkor a függvény visszatérési értéke az első, ha `Backward` akkor az utolsó illeszkedés kezdőindexe. Ha a minta nem illeszkedik a szövegre, akkor a visszaadott érték 0. Ha a `Pattern` paraméter a null sztring, akkor `Pattern_Error` kivételt vált ki.

```
function Index (Source : in String;
      Set  : in Maps.Character_Set;
      Test : in Membership := Inside;
      Going  : in Direction  := Forward)
 return Natural;
```

Ez az `Index` függvény `Test=Inside` esetén a `Set` halmaz, `Test=OutSide` esetén pedig a `Set` halmaz komplementerének egy elemét keresi a `Source` sztringben. A keresés irányát itt is a `Going` paraméter határozza meg. `Forward` esetén a legkisebb, `Backward` esetén pedig a legnagyobb indexet adja vissza. Ha nem talál megfelelő karaktert, akkor 0 lesz a visszatérési érték.

```
function Index_Non_Blank (Source : in String;
          Going  : in Direction := Forward)
 return Natural;
```

Az `Index(Source, Maps.To_Set(Space), Outside, Going)` függvényhívással ekvivalens.

```
function Count (Source : in String;
      Pattern  : in String;
      Mapping  : in Maps.Character_Mapping
          := Maps.Identity)
 return Natural;

function Count (Source : in String;
      Pattern  : in String;
      Mapping  : in Maps.Character_Mapping_Function)
 return Natural;
```

A fenti két függvény a legnagyobb olyan számot adja vissza, mely megmutatja hány nem átfedő illesztése lehetséges a `Pattern` mintának a `Source` szövegre a `Mapping` leképezésen keresztül. Azért használjuk a specifikációban a legnagyobb kifejezést, mert az átfedés meg nem engedése miatt ez a szám nem egyértelmű. Ha a `Pattern` minta null sztring, akkor a függvények `Pattern_Error` kivételt váltanak ki.

```
function Count (Source : in String;
      Set  : in Maps.Character_Set)
 return Natural;
```

Ez a `Count` függvény visszaadja, hogy hány olyan karakter van `Source`-ban, amely benne van a `Set` halmazban.

```
procedure Find_Token (Source : in String;
        Set  : in Maps.Character_Set;
        Test : in Membership;
        First  : out Positive;
        Last : out Natural);
```

A `Find_Token` eljárás a `First` és `Last` kimenő paraméterben visszaadja `Source` első olyan szeletének a kezdő- illetve végindexét, amelynek minden eleme teljesíti a `Test` paraméterben megadott feltételt a `Set` halmazra vonatkozóan úgy, hogy a közvetlenül a szelet előtti és utáni elemek (ha léteznek), nem teljesítik a feltételt. Ha nincs ilyen szelet, akkor `Last` értéke 0, `First` értéke pedig `Source'First` lesz.

```
function Translate (Source  : in String;
        Mapping : in Maps.Character_Mapping)
 return String;
function Translate (Source  : in String;
        Mapping : in Maps.Character_Mapping_Function)
 return String;
```

A fenti két függvény olyan sztringet ad vissza, amelynek hossza megegyezik `Source` hosszával, kezdőindexe 1 és minden eleme `Source` megfelelő elemének a `Mapping` leképezés szerinti képe.

```
procedure Translate (Source  : in out String;
       Mapping : in Maps.Character_Mapping);

procedure Translate (Source  : in out String;
       Mapping : in Maps.Character_Mapping_Function);
```

A fenti két eljárás a megelőző függvények eljárás változata, a leképezést `Source` összes elemére helyben végzik el.

```
function Replace_Slice (Source : in String;
        Low  : in Positive;
        High : in Natural;
        By : in String)
 return String;
```

Ha `Low > Source'Last+1` vagy `High < Source'First-1`, akkor a függvény `Index_Error` kivételt vált ki. Egyébként `High >= Low` esetén a függvényérték a `Source(Source'First..Low-1) & Source(High+1..Source'Last)`, konkatenációt foglalja magába, míg ellenkező esetben a függvényhívás ekvivalens az `Insert(Source, Before=>Low,  New_Item=>By)` hívással.

```
procedure Replace_Slice (Source : in out String;
        Low  : in Positive;
        High : in Natural;
        By : in String;
        Drop : in Truncation := Error;
        Justify  : in Alignment  := Left;
        Pad  : in Character  := Space);
```

A fenti függvény eljárás formája, a `Move(Replace_Slice(Source, Low, High, By), Source, Drop, Justify, Pad)` hívással ekvivalens.

```
function Insert (Source : in String;
       Before : in Positive;
       New_Item : in String)
 return String;
```

Ha `Before` nem esik bele a `Source'First .. Source'Last+1` intervallumba, akkor a függvény hívása `Index_Error` kivételt vált ki. Ellenkező esetben a visszatérési érték az alábbi konkatenáció 1-es kezdőindexszel:

```
Source(Source'First..Before-1) & New_Item  &
                                Source(Before..Source'Last).
```

```
procedure Insert (Source : in out String;
      Before : in Positive;
      New_Item : in String;
      Drop : in Truncation := Error);
```

Az előző függvény eljárás formája, hatása a `Move(Insert(Source, Before, New_Item), Source, Drop)` hívással ekvivalens.

```
function Overwrite (Source : in String;
        Position : in Positive;
        New_Item : in String)
 return String;
```

Ha `Position` nem esik bele a `Source'First .. Source'Last+1` intervallumba, akkor a függvény hívása `Index_Error` kivételt vált ki. Egyébként egy olyan sztringet ad vissza, amelyet `Source`-ból úgy kaphatunk meg, hogy benne `Position`-tól kezdve minden karaktert kicserélünk `New_Item` megfelelő elemére. Ha a függvény `Source` végét előbb eléri, mintsem, hogy `New_Item` elfogyna, a maradék karaktereket hozzáfűzi a sztring végéhez.

```
procedure Overwrite (Source : in out String;
       Position : in Positive;
       New_Item : in String;
       Drop : in Truncation := Right);
```

Az előző függvény eljárás formája, a `Move(Overwrite(Source, Position, New_Item), Source, Drop)` hívással ekvivalens.

```
function Delete (Source  : in String;
       From  : in Positive;
       Through : in Natural) return String;
```

```
procedure Delete (Source  : in out String;
      From  : in Positive;
      Through : in Natural;
      Justify : in Alignment := Left;
      Pad : in Character := Space);
```

`From <= Through` esetén, a visszaadott sztring `Replace_Slice(Source, From, Through, "")`, ellenkező esetben `Source`. Az eljárás változat ekvivalens a `Move(Delete(Source, From, Through), Source, Justify => Justify, Pad => Pad)` hívással.

```
function Trim (Source : in String;
     Side : in Trim_End)
  return String;

procedure Trim (Source  : in out String;
      Side  : in Trim_End;
      Justify : in Alignment := Left;
      Pad : in Character := Space);
```

A függvény változat visszatérési értékét úgy kapjuk, hogy `Side` értékétől függően `Source` elejéről (`Side = Left`), végéről (`Side = Right`) vagy mindkét végéről (`Side = Both`) elhagyjuk a `Space` karaktereket. Az eljárás változat a `Move(Trim(Source, Side), Source, Justify => Justify, Pad => Pad)` hívással ekvivalens.

```
function Trim (Source : in String;
     Left : in Maps.Character_Set;
     Right  : in Maps.Character_Set)
   return String;

procedure Trim (Source  : in out String;
      Left  : in Maps.Character_Set;
      Right : in Maps.Character_Set;
      Justify : in Alignment := Strings.Left;
      Pad : in Character := Space);
```

A függvény változat egy olyan sztringet ad vissza, amit `Source`-ból úgy kaphatunk, hogy elejéről minden `Left`-ben található karaktert, míg végéről minden `Right`-beli karaktert elhagyunk. Az eljárás változat ekvivalens a `Move(Trim(Source, Left, Right), Source, Justify => Justify, Pad => Pad)` hívással.

```
function Head (Source : in String;
     Count  : in Natural;
     Pad   : in Character := Space)
 return String;

procedure Head (Source  : in out String;
      Count : in Natural;
      Justify : in Alignment := Left;
      Pad : in Character := Space);
```

A függvény egy `Count` hosszúságú sztringet ad vissza. Ha `Count` nem nagyobb mint `Source` hossza, akkor a visszatérési értéket `Source` első `Count` darab karaktere alkotja, míg ellenkező esetben `Count-Source'Length Pad` karaktert konkatenál `Source` mögé. Az eljárás változat ekvivalens a `Move( Head(Source, Count, Pad), Source, Drop=>Error, Justify=>Justify, Pad=>Pad)` hívással.

```
function Tail (Source : in String;
     Count  : in Natural;
     Pad   : in Character := Space)
 return String;

procedure Tail (Source  : in out String;
      Count : in Natural;
      Justify : in Alignment := Left;
      Pad : in Character := Space);
```

A függvény egy `Count` hosszúságú sztringet ad vissza. Ha `Count` nem nagyobb mint `Source` hossza, akkor a visszatérési értéket `Source` utolsó `Count` darab karaktere alkotja, míg ellenkező esetben `Source` elé `Count-Source'Length` `Pad` karaktert konkatenál. Az eljárás hívása a `Move( Tail( Source, Count, Pad), Source, Drop=>Error, Justify=>Justify, Pad=>Pad)` hívással ekvivalens.

```
function "*" (Left  : in Natural;
      Right : in Character) return String;

function "*" (Left  : in Natural;
      Right : in String) return String;
```

A fenti két operátor a megadott számszor önmagával összekonkatenálja a megadott karaktert vagy sztringet. `Left = 0` esetén a visszatérési érték null sztring.

```
end Ada.Strings.Fixed;
```

Megjegyezzük még, hogy bármely olyan sztringkezelő függvény, amelynek van Character_Mapping_Function típusú paramétere, abban az esetben, ha ez a paraméter null, Constraint_Error kivételt vált ki.

A.3.2. Korlátos hosszúságú sztringek kezelése

A nyelv által definiált Strings.Bounded csomag egy sablon csomagot tartalmaz, amelynek példányai a átlátszatlan Bounded_String típust valósítják meg a hozzá tartozó műveletekkel. Egy Bounded_String típus egy objektuma egy olyan sztringet reprezentál, amelynek kezdőindexe 1, és hossza 0, illetve a sablon paramétereként a példányosításkor megadott legnagyobb méret között változhat. Mivel a Bounded_String típus átlátszatlan, a szükséges konstruktor és lekérdező műveletek is megtalálhatóak a csomagban. A beágyazott sablon csomag neve Generic_Bounded_Length:

```
with Ada.Strings.Maps;
package Ada.Strings.Bounded is
 pragma Preelaborate(Bounded);

 generic
  Max : Positive;  -- Maximum length of a Bounded_String
 package Generic_Bounded_Length is

  Max_Length : constant Positive := Max;

  type Bounded_String is private;

  Null_Bounded_String : constant Bounded_String;

  subtype Length_Range is Natural range 0 .. Max_Length;
```

A Null_Bounded_String konstans a null sztringet reprezentálja. Ha egy Bounded_String típusú objektumot másképp nem inicializálunk, akkor kezdeti értéke megegyezik a Null_Bounded_String konstanssal.

```
  function Length (Source : in Bounded_String)
     return Length_Range;
```

A Length függvény visszaadja a Source objektum által reprezentált sztring hosszát.

```
  function To_Bounded_String (Source : in String;
            Drop : in Truncation := Error)
   return Bounded_String;
```

A To_Bounded_String konstruktor egy Bounded_String objektumot hoz létre a paraméter sztringből. Ha Source'Length <= Max_Length, akkor a visszaadott objektum a Source sztringet reprezentálja. Ellenkező esetben a függvény viselkedése a Drop argumentumtól függ:

– ha Drop = Left, akkor a visszaadott Bounded_String objektum a Source sztring utolsó Max_Length karakteréből álló sztringet reprezentálja.

– ha Drop = Right, akkor a visszaadott Bounded_String objektum a Source sztring első Max_Length karakteréből álló sztringet reprezentálja.

– ha Drop = Error, akkor a függvényhívás Strings.Length_Error kivételt vált ki.

```
function To_String (Source : in Bounded_String)
  return String;
```

A To_String függvény visszaadja a paraméter által reprezentált sztringet. A függvényérték kezdőindexe 1. Az algebrai típusspecifikációhoz hasonlóan definiálva: B = To_Bounded_String(To_String(B)), ahol B egy Bounded_String.

```
function Append (Left, Right : in Bounded_String;
       Drop   : in Truncation   := Error)
 return Bounded_String;

function Append (Left   : in Bounded_String;
       Right : in String;
       Drop  : in Truncation := Error)
 return Bounded_String;

function Append (Left   : in String;
       Right : in Bounded_String;
       Drop  : in Truncation := Error)
 return Bounded_String;

function Append (Left   : in Bounded_String;
       Right : in Character;
       Drop  : in Truncation := Error)
 return Bounded_String;

function Append (Left   : in Character;
       Right : in Bounded_String;
       Drop  : in Truncation := Error)
 return Bounded_String;
```

Mindegyik `Append` függvény egy olyan `Bounded_String` objektumot ad vissza, amelyet úgy kaphatunk meg, hogy az argumentumként megadott sztringek, ill. karakterek – `Bounded_String` paraméter esetén a reprezentált sztringet tekintve – konkatenációjára alkalmazzuk a `To_Bounded_String` függvényt a függvényargumentumban megadott `Drop` értékkel.

```
procedure Append (Source : in out Bounded_String;
      New_Item : in Bounded_String;
      Drop : in Truncation  := Error);

procedure Append (Source : in out Bounded_String;
      New_Item : in String;
      Drop : in Truncation  := Error);

procedure Append (Source : in out Bounded_String;
      New_Item : in Character;
      Drop : in Truncation  := Error);
```

Mindegyik `Append(Source, New_Item, Drop)` eljárás meghívása a megfelelő `Source := Append(Source, New_Item, Drop)` értékadással ekvivalens.

```
function "&" (Left, Right : in Bounded_String)
 return Bounded_String;

function "&" (Left : in Bounded_String; Right : in String)
 return Bounded_String;

function "&" (Left : in String; Right : in Bounded_String)
 return Bounded_String;

function "&" (Left : in Bounded_String;
             Right : in Character)
 return Bounded_String;

function "&" (Left : in Character;
             Right : in Bounded_String)
 return Bounded_String;
```

Az infix `&` operátoroknak ugyanaz a hatása, mint az illeszkedő `Append` függvények meghívásának `Drop = Error` paraméterrel.

```
function Element (Source : in Bounded_String;
      Index  : in Positive)
 return Character;
```

Az `Element` függvény visszaadja a `Source` által reprezentált sztring `Index`-edik pozícióján levő karktert. Amennyiben `Index > Length(Source)`, a függvényhívás `Index_Error` kivételt vált ki.

```
procedure Replace_Element (Source : in out Bounded_String;
          Index  : in Positive;
          By : in Character);
```

A `Source` által reprezentált sztring `Index`-edik elemét kicseréli a `By` karakterre. Amennyiben `Index > Length(Source)`, az eljáráshívás `Index_Error` kivételt vált ki.

```
function Slice (Source : in Bounded_String;
      Low  : in Positive;
      High : in Natural)
 return String;
```

Visszaadja a `Source` által reprezentált sztring `Low` és `High` indexek közé eső szeletét. Ha `Low > Length(Source)+1`, akkor `Index_Error` kivételt vált ki.

```
function "="  (Left, Right : in Bounded_String)
  return Boolean;
function "="  (Left : in Bounded_String; Right : in String)
  return Boolean;

function "="  (Left : in String; Right : in Bounded_String)
  return Boolean;

function "<"  (Left, Right : in Bounded_String)
  return Boolean;

function "<"  (Left : in Bounded_String; Right : in String)
  return Boolean;

function "<"  (Left : in String; Right : in Bounded_String)
  return Boolean;

function "<=" (Left, Right : in Bounded_String)
  return Boolean;

function "<="  (Left : in Bounded_String;
  Right : in String)
  return Boolean;
```

```
function "<=" (Left : in String;
  Right : in Bounded_String)
  return Boolean;

function ">" (Left, Right : in Bounded_String)
  return Boolean;

function ">" (Left : in Bounded_String; Right : in String)
  return Boolean;

function ">" (Left : in String; Right : in Bounded_String)
  return Boolean;

function ">=" (Left, Right : in Bounded_String)
  return Boolean;

function ">=" (Left : in Bounded_String;
  Right : in String)
  return Boolean;

function ">=" (Left : in String;
  Right : in Bounded_String)
  return Boolean;
```

A fenti összehasonlító műveletek visszatérési értéke megegyezik a megfelelő `String` művelet eredményével, ha azt a paraméterként megadott sztringre – `Bounded_String` argumentum esetén az általa reprezentált sztringre – alkalmazzuk.

```
function Index (Source : in Bounded_String;
      Pattern  : in String;
      Going   : in Direction := Forward;
      Mapping  : in Maps.Character_Mapping
         := Maps.Identity)
 return Natural;

function Index (Source : in Bounded_String;
      Pattern  : in String;
      Going   : in Direction := Forward;
      Mapping  : in Maps.Character_Mapping_Function)
 return Natural;

function Index (Source : in Bounded_String;
      Set   : in Maps.Character_Set;
```

```
      Test : in Membership := Inside;
      Going  : in Direction  := Forward)
 return Natural;

function Index_Non_Blank (Source : in Bounded_String;
          Going  : in Direction := Forward)
 return Natural;

function Count (Source : in Bounded_String;
      Pattern  : in String;
      Mapping  : in Maps.Character_Mapping
          := Maps.Identity)
 return Natural;

function Count (Source : in Bounded_String;
      Pattern  : in String;
      Mapping  : in Maps.Character_Mapping_Function)
 return Natural;

function Count (Source : in Bounded_String;
      Set  : in Maps.Character_Set)
 return Natural;

procedure Find_Token (Source : in Bounded_String;
        Set  : in Maps.Character_Set;
        Test : in Membership;
        First  : out Positive;
        Last : out Natural);
```

A fenti kereső eljárások és függvények hatása megegyezik a `Strings.Fixed` csomag megfelelő eljárásának, ill. függvényének a `Bounded_String` paraméter által reprezentált sztringre történő meghívásával.

```
function Translate (Source  : in Bounded_String;
        Mapping : in Maps.Character_Mapping)
 return Bounded_String;

procedure Translate (Source  : in out Bounded_String;
       Mapping : in Maps.Character_Mapping);

function Translate (Source  : in Bounded_String;
        Mapping : in Maps.Character_Mapping_Function)
 return Bounded_String;
```

```
procedure Translate (Source  : in out Bounded_String;
       Mapping : in Maps.Character_Mapping_Function);
```

A `Translate` függvények és eljárások hatása a megfelelő `Strings.Fixed` csomagbeli alprogrammal analóg.

```
function Replace_Slice (Source : in Bounded_String;
        Low  : in Positive;
        High : in Natural;
        By : in String;
        Drop : in Truncation := Error)
 return Bounded_String;

procedure Replace_Slice (Source : in out Bounded_String;
         Low  : in Positive;
         High : in Natural;
         By : in String;
         Drop : in Truncation := Error);

function Insert (Source : in Bounded_String;
       Before : in Positive;
       New_Item : in String;
       Drop : in Truncation := Error)
 return Bounded_String;

procedure Insert (Source : in out Bounded_String;
      Before : in Positive;
      New_Item : in String;
      Drop : in Truncation := Error);

function Overwrite (Source  : in Bounded_String;
        Position  : in Positive;
        New_Item  : in String;
        Drop  : in Truncation := Error)
 return Bounded_String;

procedure Overwrite (Source  : in out Bounded_String;
       Position  : in Positive;
       New_Item  : in String;
       Drop  : in Truncation := Error);

function Delete (Source  : in Bounded_String;
       From  : in Positive;
       Through : in Natural) return Bounded_String;
```

```
procedure Delete (Source  : in out Bounded_String;
      From    : in Positive;
      Through : in Natural);

function Trim (Source : in Bounded_String;
     Side : in Trim_End)
 return Bounded_String;

procedure Trim (Source : in out Bounded_String;
      Side : in Trim_End);

function Trim (Source : in Bounded_String;
     Left : in Maps.Character_Set;
     Right  : in Maps.Character_Set)
 return Bounded_String;

procedure Trim (Source : in out Bounded_String;
      Left : in Maps.Character_Set;
      Right  : in Maps.Character_Set);

function Head (Source : in Bounded_String;
     Count  : in Natural;
     Pad  : in Character  := Space;
     Drop : in Truncation := Error)
 return Bounded_String;

procedure Head (Source : in out Bounded_String;
      Count  : in Natural;
      Pad  : in Character  := Space;
      Drop : in Truncation := Error);

function Tail (Source : in Bounded_String;
     Count  : in Natural;
     Pad  : in Character  := Space;
     Drop : in Truncation := Error)
 return Bounded_String;

procedure Tail (Source : in out Bounded_String;
      Count  : in Natural;
      Pad  : in Character  := Space;
      Drop : in Truncation := Error);
```

A transzformációs és lekérdező alprogramok hatása a `Strings.Fixed` csomag megfelelő eljárásán, ill. függvényén alapszik. Mindegyik alprogram esetén az eredményt úgy kapjuk, hogy a megfelelő `Strings.Fixed`-beli eljárást alkalmazzuk a `Bounded_String` paraméter által reprezentált sztringre, majd a `To_Bounded_String` műveletet alkalmazzuk az eredményre `Drop` második paraméterrel.

```
   function "*" (Left  : in Natural;
         Right : in Character)
    return Bounded_String;

   function "*" (Left  : in Natural;
         Right : in String)
    return Bounded_String;

   function "*" (Left  : in Natural;
         Right : in Bounded_String)
    return Bounded_String;

   function Replicate (Count : in Natural;
           Item  : in Character;
           Drop  : in Truncation := Error)
    return Bounded_String;

   function Replicate (Count : in Natural;
           Item  : in String;
           Drop  : in Truncation := Error)
    return Bounded_String;

   function Replicate (Count : in Natural;
           Item  : in Bounded_String;
           Drop  : in Truncation := Error)
    return Bounded_String;

  private
    ...
  end Generic_Bounded_Length;

 end Ada.Strings.Bounded;
```

A `"*"` infix operátor és a `Replicate` függvény is a `Strings.Fixed`-beli `"*"` műveleten alapszik, a transzformációs műveletekhez hasonló módon. A `"*"` művelet `Strings.Length_Error` kivételt vált ki, ha az eredmény hossza túllépné `Max_Length` értékét.

A.3.3. Nemkorlátos hosszúságú sztringek kezelése

A szabványos `Strings.Unbounded` csomag az `Unbounded_String` átlátszatlan típust és a műveleteit definiálja. Az `Unbounded_String` típus egy olyan sztringet reprezentál, amelynek kezdőindexe 1, hossza pedig elméletileg 0 és `Natural'Last` között változik. A `Strings.Unbounded` csomag deklarációja:

```
with Ada.Strings.Maps;
package Ada.Strings.Unbounded is
 pragma Preelaborate(Unbounded);

 type Unbounded_String is private;

 Null_Unbounded_String : constant Unbounded_String;

 function Length (Source : in Unbounded_String)
    return Natural;

 type String_Access is access all String;
 procedure Free (X : in out String_Access);
```

A `Null_Unbounded_String` konstans a null sztringet reprezentálja. Ha egy `Unbounded_String` típusú objektumot másként nem inicializálunk, akkor értéke megegyezik a `Null_Unbounded_String` konstans értékével. A `Length` függvény visszaadja a reprezentált sztring hosszát.

A `String_Access` típus lehetőséget ad a nemkorlátos hosszúságú sztringek explicit kezelésére. A `Free` eljárás elvégzi a kapott objektum ellenőrizetlen felszabadítását (unchecked deallocation).

```
 function To_Unbounded_String (Source : in String)
   return Unbounded_String;

 function To_Unbounded_String (Length : in Natural)
   return Unbounded_String;

 function To_String (Source : in Unbounded_String)
   return String;
```

A fenti konverziós függvények közül az első egy olyan `Unbounded_ String` objektumot ad vissza, amely a `Source` paramétert reprezentálja. A második függvény egy inicializálatlan tartalmú, a `Length` paraméterben megadott hosszúságú sztringet reprezentáló objektumot ad vissza. A fordított irányú konverzió a reprezentált sztringet adja vissza 1-es kezdőindexszel.

```
procedure Append (Source : in out Unbounded_String;
      New_Item : in Unbounded_String);

procedure Append (Source : in out Unbounded_String;
      New_Item : in String);

procedure Append (Source : in out Unbounded_String;
      New_Item : in Character);
```

Mindegyik Append művelet eredményeként a Source paraméter által reprezentált sztring, az eredeti érték és a New_Item paraméter konkatenációja lesz.

```
function "&" (Left, Right : in Unbounded_String)
  return Unbounded_String;

function "&" (Left : in Unbounded_String;
  Right : in String)
  return Unbounded_String;

function "&" (Left : in String;
  Right : in Unbounded_String)
  return Unbounded_String;

function "&" (Left : in Unbounded_String;
  Right : in Character)
  return Unbounded_String;

function "&" (Left : in Character;
  Right : in Unbounded_String)
  return Unbounded_String;
```

A fenti "&" infix operátorok visszatérési értéke egy olyan Unbounded_String objektum lesz, amely a két paraméter konkatenációját reprezentálja (ha a bejövő paraméter Unbounded_String típusú, akkor természetesen az általa reprezentált sztringet kell konkatenálni).

```
function Element (Source : in Unbounded_String;
      Index  : in Positive)
 return Character;

procedure Replace_Element (Source : in out Unbounded_String;
         Index  : in Positive;
         By : in Character);
```

```
function Slice (Source : in Unbounded_String;
      Low  : in Positive;
      High : in Natural)
 return String;
```

A fenti három alprogram hatása megegyezik a megfelelő `Bounded_String` művelet hatásával.

```
function "="  (Left, Right : in Unbounded_String)
  return Boolean;

function "="  (Left : in Unbounded_String;
  Right : in String)
  return Boolean;

function "="  (Left : in String;
  Right : in Unbounded_String)
  return Boolean;

function "<"  (Left, Right : in Unbounded_String)
  return Boolean;

function "<"  (Left : in Unbounded_String;
  Right : in String)
  return Boolean;

function "<"  (Left : in String; Right :
  in Unbounded_String)
  return Boolean;

function "<=" (Left, Right : in Unbounded_String)
  return Boolean;

function "<="  (Left : in Unbounded_String;
  Right : in String)
  return Boolean;

function "<="  (Left : in String;
  Right : in Unbounded_String)
  return Boolean;

function ">"  (Left, Right : in Unbounded_String)
  return Boolean;
```

```
function ">"  (Left : in Unbounded_String;
  Right : in String)
  return Boolean;

function ">"  (Left : in String;
  Right : in Unbounded_String)
  return Boolean;

function ">=" (Left, Right : in Unbounded_String)
  return Boolean;

function ">="  (Left : in Unbounded_String;
  Right : in String)
  return Boolean;

function ">="  (Left : in String;
  Right : in Unbounded_String)
  return Boolean;
```

A fenti összehasonlító operátorok eredménye megegyezik a megfelelő `String` operátorok eredményével, ha az `Unbounded_String` típusú paraméterek helyett az azok által reprezentált sztringet tekintjük.

```
function Index (Source : in Unbounded_String;
      Pattern  : in String;
      Going   : in Direction := Forward;
      Mapping  : in Maps.Character_Mapping
           := Maps.Identity)
 return Natural;

function Index (Source : in Unbounded_String;
      Pattern  : in String;
      Going   : in Direction := Forward;
      Mapping  : in Maps.Character_Mapping_Function)
 return Natural;

function Index (Source : in Unbounded_String;
      Set  : in Maps.Character_Set;
      Test : in Membership := Inside;
      Going  : in Direction  := Forward) return Natural;

function Index_Non_Blank (Source : in Unbounded_String;
      Going  : in Direction := Forward) return Natural;
```

```
function Count (Source : in Unbounded_String;
      Pattern  : in String;
      Mapping  : in Maps.Character_Mapping
           := Maps.Identity)
 return Natural;

function Count (Source : in Unbounded_String;
      Pattern  : in String;
      Mapping  : in Maps.Character_Mapping_Function)
 return Natural;

function Count (Source : in Unbounded_String;
      Set  : in Maps.Character_Set)
 return Natural;

procedure Find_Token (Source : in Unbounded_String;
        Set  : in Maps.Character_Set;
        Test : in Membership;
        First  : out Positive;
        Last : out Natural);
```

A fenti kereső műveletek ugyanúgy működnek, mint a `Strings.Fixed` csomagbeli párjuk, ha az `Unbounded_String` típusú paraméterek helyett az azok által reprezentált sztringet vesszük figyelembe.

```
function Translate (Source   : in Unbounded_String;
        Mapping : in Maps.Character_Mapping)
 return Unbounded_String;

procedure Translate (Source   : in out Unbounded_String;
       Mapping : in Maps.Character_Mapping);

function Translate (Source   : in Unbounded_String;
        Mapping : in Maps.Character_Mapping_Function)
 return Unbounded_String;

procedure Translate (Source   : in out Unbounded_String;
       Mapping : in Maps.Character_Mapping_Function);
```

A leképező műveletek analógok a `Strings.Fixed`-beli párjukkal: a leképezést alkalmazzuk az `Unbounded_String` paraméter által reprezentált sztringre, majd az eredményt – a `To_Unbounded_String` művelettel – visszakonvertáljuk `Unbounded_String`-gé.

```
function Replace_Slice (Source : in Unbounded_String;
        Low  : in Positive;
        High : in Natural;
        By : in String)
 return Unbounded_String;

procedure Replace_Slice (Source : in out Unbounded_String;
         Low  : in Positive;
         High : in Natural;
         By : in String);

function Insert (Source : in Unbounded_String;
       Before : in Positive;
       New_Item : in String)
 return Unbounded_String;

procedure Insert (Source : in out Unbounded_String;
      Before : in Positive;
      New_Item : in String);

function Overwrite (Source  : in Unbounded_String;
        Position  : in Positive;
        New_Item  : in String)
 return Unbounded_String;

procedure Overwrite (Source  : in out Unbounded_String;
       Position  : in Positive;
       New_Item  : in String);

function Delete (Source  : in Unbounded_String;
       From  : in Positive;
       Through : in Natural)
 return Unbounded_String;

procedure Delete (Source  : in out Unbounded_String;
      From  : in Positive;
      Through : in Natural);

function Trim (Source : in Unbounded_String;
     Side : in Trim_End)
 return Unbounded_String;

procedure Trim (Source : in out Unbounded_String;
      Side : in Trim_End);
```

```
  function Trim (Source : in Unbounded_String;
       Left : in Maps.Character_Set;
       Right  : in Maps.Character_Set)
   return Unbounded_String;

  procedure Trim (Source : in out Unbounded_String;
        Left : in Maps.Character_Set;
        Right  : in Maps.Character_Set);

  function Head (Source : in Unbounded_String;
       Count  : in Natural;
       Pad  : in Character := Space)
   return Unbounded_String;

  procedure Head (Source : in out Unbounded_String;
        Count  : in Natural;
        Pad  : in Character := Space);

  function Tail (Source : in Unbounded_String;
       Count  : in Natural;
       Pad  : in Character := Space)
   return Unbounded_String;

  procedure Tail (Source : in out Unbounded_String;
        Count  : in Natural;
        Pad  : in Character := Space);

  function "*" (Left  : in Natural;
        Right : in Character)
   return Unbounded_String;

  function "*" (Left  : in Natural;
        Right : in String)
   return Unbounded_String;

  function "*" (Left  : in Natural;
        Right : in Unbounded_String)
   return Unbounded_String;

private
 ...
end Ada.Strings.Unbounded;
```

A "*" operátor, valamint a transzformációs és lekérdező függvények az előzőkhöz hasonlóan analógok a `Strings.Fixed`-beli párjukkal. Minden esetben a fix hosszú műveletet alkalmazzuk a reprezentált sztringre, majd a `To_Unbounded_String` művelettel `Unbounded_String` objektumot csinálunk az eredményből. Az eljárások végrehajtásának eredményeként a `Source` paraméter által reprezentált sztring meg fog egyezni a fixhosszú sztringekre vonatkozó megfelelő függvény értékével.

A.3.4. Sztringkezelést támogató halmazok és leképezések

A fenti sztringkezelő csomagok támaszkodnak a `Strings.Maps` csomagban definiált halmaz és leképezés típusokra. A műveletek egyszerű felhasználását támogatja az `Ada.Strings.Maps.Constants` csomag, amely a gyakran használt halmazokat és leképezéseket konstansként deklarálja.

```
package Ada.Strings.Maps.Constants is
 pragma Preelaborate(Constants);

 Control_Set : constant Character_Set;
 Graphic_Set : constant Character_Set;
 Letter_Set  : constant Character_Set;
 Lower_Set : constant Character_Set;
 Upper_Set : constant Character_Set;
 Basic_Set : constant Character_Set;
 Decimal_Digit_Set : constant Character_Set;
 Hexadecimal_Digit_Set : constant Character_Set;
 Alphanumeric_Set  : constant Character_Set;
 Special_Set : constant Character_Set;
 ISO_646_Set : constant Character_Set;

 Lower_Case_Map  : constant Character_Mapping;
 Upper_Case_Map  : constant Character_Mapping;
 Basic_Map : constant Character_Mapping;
private
  ...
end Ada.Strings.Maps.Constants;
```

A fent definiált halmazok ugyanannak az osztályozásnak felelnek meg, mint amit a `Characters.Handling` csomagban vezettünk be. A `Lower_Case_Map` a nagybetűket kicsivé konvertálja, a többit pedig helybenhagyja. Analóg módon az `Upper_Case_Map` a kisbetűket naggyá konvertálja, a többit pedig helybenhagyja. A `Basic_Map` leképezés az ékezetes betűkről elhagyja az ékezeteket, míg a többi karaktert helybenhagyja.

A.3.5. Wide_String kezelés

A Wide_Character elemekből álló sztringek kezelése a Strings.Wide_Maps, Strings.Wide_Fixed, Strings.Wide_Bounded, Strings.Wide_Unbounded és Strings.Wide_Maps.Wide_Constants csomagokon keresztül lehetséges. Ezek a csomagok ugyanazokat a műveleteket biztosítják mint a Character elemekből álló sztringekre vonatkozó párjaik. Ez utóbbi csomag tartalmazza még a

```
Character_Set : constant Wide_Maps.Wide_Character_Set;
```

konstans deklarációját, amely halmaz pontosan azokat a Wide_Character elemeket tartalmazza, amelyekre a Characters.Is_Character függvény értéke True.

A.4. A numerikus csomagok

A Numerics könyvtári csomag a szülője több olyan csomagnak, amelyekkel a nyelv a matematikai számításokat támogatja:

```
package Ada.Numerics is
 pragma Pure(Numerics);
 Argument_Error : exception;
 Pi : constant :=
    3.14159_26535_89793_23846_26433_
      83279_50288_41971_69399_37511;
 e  : constant :=
    2.71828_18284_59045_23536_02874_
      71352_66249_77572_47093_69996;
end Ada.Numerics;
```

Az Argument_Error kivételt a gyerek csomagok műveletei akkor váltják ki, ha a művelet valamelyik argumentuma kívül esik a megfelelő matematikai függvény értelmezési tartományán. Az egyes implementációk definiálhatják a Pi és e konstansok értékét nagyobb pontossággal is.

A.4.1. Elemi függvények

Az elemi függvényként ismert matematikai függvények közelítő implementációit a Numerics. Generic_Elementary_Functions csomag tartalmazza. Ugyanennek a csomagnak van nem sablon változata is az összes előre definiált lebegőpontos típusra. Ezek a csomagok is a Numerics csomagból vannak származtatva. Az alprogramok pontossága implementációfüggő.

A Numerics.Generic_Elementary_Functions csomag deklarációja:

```
generic
 type Float_Type is digits <>;
package Ada.Numerics.Generic_Elementary_Functions is
 pragma Pure(Generic_Elementary_Functions);

 function Sqrt  (X : Float_Type'Base)
   return Float_Type'Base;
 function Log (X : Float_Type'Base)  return Float_Type'Base;
 function Log (X, Base : Float_Type'Base)
   return Float_Type'Base;
 function Exp (X : Float_Type'Base)  return Float_Type'Base;
 function "**"  (Left, Right : Float_Type'Base)
    return Float_Type'Base;

 function Sin (X : Float_Type'Base)  return Float_Type'Base;
 function Sin (X, Cycle  : Float_Type'Base)
    return Float_Type'Base;
 function Cos (X : Float_Type'Base)  return Float_Type'Base;
 function Cos (X, Cycle  : Float_Type'Base)
    return Float_Type'Base;
 function Tan (X : Float_Type'Base)  return Float_Type'Base;
 function Tan (X, Cycle  : Float_Type'Base)
    return Float_Type'Base;
 function Cot (X : Float_Type'Base)  return Float_Type'Base;
 function Cot (X, Cycle  : Float_Type'Base)
    return Float_Type'Base;

 function Arcsin  (X : Float_Type'Base)
    return Float_Type'Base;
 function Arcsin  (X, Cycle  : Float_Type'Base)
    return Float_Type'Base;
 function Arccos  (X : Float_Type'Base)
    return Float_Type'Base;
 function Arccos  (X, Cycle  : Float_Type'Base)
    return Float_Type'Base;
 function Arctan  (Y : Float_Type'Base;
                   X   : Float_Type'Base := 1.0)
    return Float_Type'Base;
 function Arctan  (Y : Float_Type'Base;
                   X   : Float_Type'Base := 1.0;
                   Cycle : Float_Type'Base)
    return Float_Type'Base;
 function Arccot  (X : Float_Type'Base;
       Y   : Float_Type'Base := 1.0) return Float_Type'Base;
```

```
function Arccot  (X : Float_Type'Base;
      Y   : Float_Type'Base := 1.0;
      Cycle : Float_Type'Base)  return Float_Type'Base;

function Sinh  (X : Float_Type'Base)
   return Float_Type'Base;
function Cosh  (X : Float_Type'Base)
   return Float_Type'Base;
function Tanh  (X : Float_Type'Base)
   return Float_Type'Base;
function Coth  (X : Float_Type'Base)
   return Float_Type'Base;
function Arcsinh (X : Float_Type'Base)
   return Float_Type'Base;
function Arccosh (X : Float_Type'Base)
   return Float_Type'Base;
function Arctanh (X : Float_Type'Base)
   return Float_Type'Base;
function Arccoth (X : Float_Type'Base)
   return Float_Type'Base;

end Ada.Numerics.Generic_Elementary_Functions;
```

A függvények a szokásos matematikai függvényeket számolják ki. Az egyargumentumú `Log` függvény a természetes alapú logaritmust szolgáltatja. A kétoperandusú `Log` függvényben a második paraméter a logaritmus alapja. Azokban a trigonometrikus függvényekben, ahol nem szerepel a `Cycle` paraméter, ott az argumentumot radiánban kell megadni, és természetesen az ugyanilyen inverz függvények eredményét is radiánban kapjuk meg. Azokban a függvényekben, ahol szerepel a `Cycle` paraméter, ott annak segítségével átskálázhatjuk a mértékegységet úgy, hogy a függvény a paraméter értéke szerint lesz periodikus.

Az `Arctan` és `Arccot` függvények kissé különböznek a matematikában megszokottól. Az `Arctan(y, x)` (ill. az `Arctan(y, x, Cycle)`) függvény értéke `x` értékétől függően az alábbi lehet:

– `Pi/2` (`Cycle/4`), ha `x = 0` és `y > 0`,

– `-Pi/2` (`-Cycle/4`), ha `x = 0` és `y < 0`,

– az a szög radiánban (a `Cycle` által meghatározott mértékegységben), amelynek tangense `y/x` és abban a síknegyedben van, amelyben az `(x,y)` pont, ha `x/=0` és `y/=0`.

Az `Arccot(x, y)` (ill. az `Arccot(x, y, Cycle)`) függvény értéke `y` értékétől függően az alábbi lehet: – `0`, ha `y = 0` és `x > 0`,

– `Pi` (`Cycle/2`), ha `y = 0` és `x < 0` (`y` előjelétől függően lehet `-Pi` is),

– az a szög radiánban (a `Cycle` által meghatározott mértékegységben), amelynek cotangense `x/y` és abban a síknegyedben van, amelyben az `(x,y)` pont, ha `x/=0` és `y/=0`.

Informálisan fogalmazva, minkét függvény azt a szöget adja vissza, amelyet az origóból kiinduló `(X, Y)` vektor, és a szintén az origóból kiinduló `(1, 0)` vektor bezár.

A matematikai műveletek kivételeket váltanak ki nem megfelelő értékkel történő hívás esetén. Az alábbi függvények `Numerics.Argument_Error` kivételt váltanak ki, ha az aktuális paraméter kívül esik a függvény értelmezési tartományán, azaz az alábbi esetekben:

– azon trigonometrikus függvények, ahol a `Cycle` paraméter szerepel, ha ez a paraméter 0 vagy negatív,

– azon logaritmusfüggvény, amelynél az alapot is meg lehet adni, negatív, 0 vagy 1 alap esetén,

– az `Sqrt` és `Log` függvény, negatív X paraméter esetén,

– a hatványozás operátor, negatív alap esetén, vagy akkor, ha mindkét operandus 0,

– az `Arcsin`, `Arccos`, és `Arctanh` függvények, ha az `X` paraméter nem esik a [-1, 1] intervallumba,

– az `Arctan` és `Arccot` függvények, ha mind az `X`, mind az `Y` paraméterek értéke 0,

– az `Arccosh` függvény, ha az `X` paramétek kisebb mint 1,

– az `Arccoth` függvény, ha az `X` paraméter beleesik a (-1, 1) nyílt intervallumba.

Az alábbi függvények `Constraint_Error` kivételt váltanak ki az alábbi esetekben, feltéve, hogy `Float_Type'Machine_Overflows` értéke `True`:

– a `Log`, `Cot` és `Coth` függvények, ha az `X` paraméter 0,

– a hatányozás operátor, ha az alap 0 és a kitevő negatív,

– azon `Tan` függvény, ahol a periódus megadható, ha az `X` paraméter páratlan többszöröse a `Cycle` paraméter negyedének,

– azon `Cot` függvény, ahol a periódus megadható, ha az `X` paraméter 0, vagy többszöröse a `Cycle` paraméter felének,

– az `Arctanh` és `Arccoth` függvények, ha az `X` argumentum abszolút értéke 1.

`Constraint_Error` kivétel váltódik ki azokban az esetekben is, amikor a függvényérték túl- vagy alulcsordul, azaz nem fér bele a számábrázolásba. Abban az esetben, ha a fenti két kivétel mindegyikének feltételei teljesülnek, akkor a `Numerics.Argument_Error` kivétel váltódik ki.

Az előredefiniált lebegőpontos típusokra vonatkozó elemi függvények nem sablon csomagjainak specifikációját úgy kaphatjuk meg, hogy a fenti csomagdefinícióban a `Float_Type'Base` típust a megfelelőre cseréljük. A csomagok nevét úgy kapjuk, hogy a fenti csomag nevében a `Generic` szót a megfelelő típus nevére cseréljük pl: `Numerics.Short_Elementary_Functions`. E

névkonvenció alól kivételt képez a `Float` típus, amely esetén a csomag neve: `Numerics.Elementary_Functions`.

A.4.2. Véletlenszámok generálása

A `Numerics.Float_Random` csomagban találhatóak a lebegőpontos véletlenszám-generátor műveletek. Ezen kívül a `Numerics.Discrete_Random` sablon csomag felhasználásával példányosíthatunk csomagokat tetszőleges egész és felsorolt típusokra. A `Numerics.Float_Random` csomag deklarációja:

```
package Ada.Numerics.Float_Random is

  type Generator is limited private;

  subtype Uniformly_Distributed is Float range 0.0 .. 1.0;
  function Random (Gen : Generator)
     return Uniformly_Distributed;
```

A `Generator` korlátozott átlátszatlan típus egy véletlenszám-sorozathoz kötődik. Minden objektumnak van egy belső állapota, amely jellemzi a kapcsolódó sorozatbeli aktuális pozíciót. Minden generátor automatikusan egy olyan nemspecifikált belső állapottal inicializálódik, ami nem változik a program egyes végrehajtásai között. Természetesen a generátor explicit inicializálható vagy újrainicializálható egy időpontfüggő, vagy egy korábban elmentett, esetleg egy megadott egész számból képzett állapotra.

A `Random` függvény a következő véletlenszámot adja vissza. A véletlenszámot a belső állapotból – annak megváltoztatásával – egy implementációfüggő algoritmus szolgáltatja. A generált szám `Uniformly_Distributed` típusú, azaz beleesik a `0.0..1.0` intervallumba.

```
  procedure Reset (Gen : in Generator;
         Initiator : in Integer);
```

Újrainicializálja a véletlenszám-generátort a megadott `Initiator` egész érték alapján. Az algoritmus, amely a generátor belső állapotát az adott szám alapján előállítja implementációnként változhat.

```
  procedure Reset (Gen : in Generator);
```

Úgynevezett időpontfüggő inicializálás. A generátor belső állapotát egy implementációfüggő algoritmussal – például a rendszeridő alapján – állítja be.

```
  type State is private;

  procedure Save  (Gen  : in  Generator;
         To_State : out State);
  procedure Reset (Gen  : in  Generator;
         From_State : in  State);
```

A `State` átlátszatlan típus a generátor belső állapotát jellemző adatokat tartalmazza. A `Save` eljárás ezt az aktuális belső állapotot adja vissza. Ez a harmadik `Reset` művelet a véletlenszám-generátor belső állapotát a `From_State` paraméter értékére állítja.

```
  Max_Image_Width : constant := implementation-defined
     integer value;

  function Image (Of_State   : State)  return String;
  function Value (Coded_State : String) return State;
 private
  ...
 end Ada.Numerics.Float_Random;
```

Az `Image` és `Value` műveletek lehetővé teszik a véletleszám-generátor egy állapotának elmentését és visszatöltését. Az `Image` művelet egy legfeljebb `Max_Image_Width` hosszúságú sztringet képez a megadott paraméterből (ezt aztán akár fájlba is lehet menteni). A `Value` függvény az `Image` művelet inverze, azaz `Value(Image(S)) = S` minden `S` generátorból nyert állapot-objektumra. A sztring képzési módja az egyes implementációkban különböző lehet. Ha a `Value` függvényt egy olyan sztringgel hívjuk meg, ami nem egy generátor állapot képe, akkor a hívás `Constraint _Error` kivételt vált ki.

A `Numerics.Discrete_Random` sablon csomag lehetőséget ad diszkrét típusú véletlen értékek generálására. A sablon paramétere a generálandó értékek típusa:

```
generic
 type Result_Subtype is (<>);
package Ada.Numerics.Discrete_Random is

 type Generator is limited private;

 function Random (Gen : Generator) return Result_Subtype;

 procedure Reset (Gen : in Generator;
        Initiator : in Integer);
```

```
  procedure Reset (Gen : in Generator);

  type State is private;

  procedure Save  (Gen   : in  Generator;
         To_State : out State);
  procedure Reset (Gen   : in  Generator;
         From_State : in  State);

  Max_Image_Width : constant := implementation-defined
     integer value;

  function Image (Of_State   : State)  return String;
  function Value (Coded_State : String) return State;

 private
  ...
 end Ada.Numerics.Discrete_Random;
```

A diszkrét típusú véletlen értékek generálását támogató csomag mind felépítésében, mind működésében analóg a lebegőpontos véletlenszám generátor csomaggal, ezért a csomag tartalmát nem részletezzük. A leírás azonban annyival kiegészül, hogy ha a csomagot egy olyan altípussal akarjuk példányosítani, amelynek értékkészlete üres, akkor `Constraint_Error` hibajelzést kapunk.

A.4.3. A lebegőpontos típusok attribútumai

Egy `T` lebegőpontos típus minden `S` altípusára definiálva vannak az alábbi reprezentáció-orientált attribútumok:

```
S'Machine_Radix
```

Visszaadja a hardver reprezentáció számrendszerének alapját. A visszaadott érték `universal_integer` típusú.

A többi attribútumot a lebegőpontos szám kanonikus alakjának segítségével definiáljuk. A `T` típus kanonikus alakja:

$+/\text{-mantissza}*\texttt{T'Machine_Radix}^{\text{kitevő}}$

ahol mantissza egy `T'Machine_Radix` alapú számrendszerbeli tört, melynek első számjegye nem 0, a kitevő pedig egész szám. Itt jegyezzük meg, hogy a 0.0 lebegőpontos értéknek nincs kanonikus alakja.

```
S'Machine_Mantissa
```

Azt a legnagyobb (`universal_integer` típusú) `p` számot adja vissza, amelyre a `T` összes `p` számjegyből álló mantisszájú, tetszőleges `T'Machine_Emin` és `T'Machine_Emax` érték közé eső kitevőjű kanonikus formában kifejezhető értéke egy a `T` típust ábrázoló gépi szám.

`S'Machine_Emin`

Azt a legkisebb (`universal_integer` típusú) `e` számot adja vissza, amelyre a `T` összes olyan kanonikus formában kifejezhető értéke, amelynek mantisszája `T'Machine_Mantissa` számjegyből áll, és kitevője `e`, egy a `T` típust ábrázoló gépi szám.

`S'Machine_Emax`

Azt a legnagyobb (`universal_integer` típusú) `e` számot adja vissza, amelyre a `T` összes olyan kanonikus formában kifejezhető értéke, amelynek mantisszája `T'Machine_Mantissa` számjegyből áll, és kitevője `e`, egy a `T` típust ábrázoló gépi szám.

`S'Denorm`

Igazat ad, ha minden az alábbi formában kifejezhető érték

+/-mantissza*`T'Machine_Radix`$^{\texttt{T'Machine_Emin}}$

ahol mantissza egy `T'Machine_Radix` alapú `T'Machine_Mantissa` számjegyű tört amelynek első jegye 0, egy a `T` típust reprezentáló gépi szám. Egyébként `False`.

Denormalizáltnak nevezünk egy lebegőpontos értéket, ha a fenti formában kifejezhető. Minden olyan nemnulla gépi számot, amely nem denormalizált, normalizált számnak nevezünk. Informálisan az `S'Denorm` `True` értéke azt jelzi, hogy a `T` típus ábrázolásában előfordul(hat)nak denormalizált számok.

`S'Machine_Rounds`

`True` értéket ad, ha minden olyan előre definiált művelet, amely `T` típusú értéket ad vissza, pontatlan eredmény esetén kerekít, míg ellenkező esetben az attribútum értéke `False`.

`S'Machine_Overflows`

`True` értéket ad, ha a túlcsordulás és a nullával történő osztás minden olyan előredefiniált műveletben, amelynek értéke `T` típusú, `Constraint_Error` kivételt vált ki. Ellenkező esetben `False`.

`S'Signed_Zeros`

`True` értéket ad, ha a `T` típus gépi reprezentációjában pozitív és negatív előjele is lehet a nulla értéknek, `False`-t egyébként.

Az alábbi primitív függvény-attribútumokat használhatjuk a `T` lebegőpontos típus minden `S` altípusára:

```
function S'Exponent (X : T) return universal_integer;
```

Visszaadja az `X` normalizált kitevőjét. A normalizált kitevő definíciója az alábbi:

- a 0 normalizált kitevője 0,
- egy nem 0 `x`-re `x` normalizált kitevője az az egyértelmű `k` szám, amelyre `T'Machine_Radix`$^{k-1}$`<= |x| < T' Machine_Radix`^{k}.

```
function S'Fraction (X : T) return T;
```

Az `X*T'Machine_Radix`^{k} értéket adja vissza, ahol `k` az `X` normalizált kitevője. A 0 visszatérési érték – amely csak `X=0` esetén lehetséges – előjele megegyezik az `X` előjelével.

```
function S'Compose (Fraction : T;
   Exponent : universal_integer) return T;
```

Legyen `v := Fraction*T'Machine_Radix`$^{\texttt{Exponent-k}}$, ahol `k` a `Fraction` paraméter normalizált kitevője. Ha `v` egy a `T` típust ábrázoló gépi szám, vagy `|v| >= T'Model_Small`, akkor a függvényérték `v` lesz. Ellenkező esetben a `v`-vel szomszédos gépi számok egyikét adja. `S'Signed_Zeros = True` és 0 eredmény esetén az előjel a `Fraction` paraméter előjele lesz. Ha `v` kívül esik az `S` típus értékkészletén. `Constraint_Error` kivételt vált ki,

```
function S'Scaling (X : T; Adjustment : universal_integer)
   return T;
```

Legyen `v := X*T'Machine_Radix`$^{\texttt{Adjustment}}$. Ha `v` egy a `T` típust ábrázoló gépi szám, vagy $|v|$ >= `T'Model_Small`, akkor a függvényérték `v` lesz. Ellenkező esetben a `v`-vel szomszédos gépi számok egyikét adja.

`S'Signed_Zeros = True` és 0 eredmény esetén az előjel az `X` paraméter előjele lesz. `Constraint_Error` kivételt válthat ki, ha `v` kívül esik az `S` típus értékkészletén.

```
function S'Floor (X : T) return T;
```

A függvény az `X` paraméter egészrészét adja vissza. Ha `X` nulla, akkor a visszaadott érték előjele megegyezik `X` előjelével. Más olyan esetekben, amikor az eredmény nulla, az előjel pozitív.

```
function S'Ceiling (X : T) return T;
```

A függvény az `X` paraméter felső egészrészét adja vissza. Ha `X` nulla, akkor a visszaadott érték előjele megegyezik `X` előjelével. Más olyan esetekben, amikor az eredmény nulla, `S'Signed_Zeros = True` esetén az előjel negatív.

```
function S'Rounding (X : T) return T;
```

`X` egészre kerekített értékét adja vissza, a kerekítés szabályának megfelelően a pontosan fél értéket felfelé kerekíti. Ha a visszaadott érték nulla, akkor `S'Signed_Zeros = True` esetén az előjele megegyezik az `X` előjelével.

```
function S'Unbiased_Rounding (X : T) return T;
```

A függvény az `X`-hez legközelebb eső egész számot adja vissza, a pontosan fél értéket pedig a szomszédos páros számra kerekíti. Ha a visszaadott érték nulla, akkor `S'Signed_Zeros = True` esetén az előjele megegyezik az `X` előjelével.

```
function S'Truncation (X : T) return T;
```

A függvény negatív `X` értékekre `Ceiling(X)`, ellenkező esetben `Floor(X)` értékét adja vissza.

```
function S'Remainder (X, Y : T) return T;
```

Nem nulla `Y` esetén legyen `v := X-n*Y`, ahol `n` az `X/Y` értékhez legközelebb eső egész (ha `X/Y` törtrésze pontosan fél, akkor a páros `n`-et választjuk). Ha `v` egy a `T` típust reprezentáló gépi szám, akkor a függvényérték `v`, máskülönben nulla. `Y=0` esetén a függvényhívás `Constraint_Error` kivételt vált ki. `S'Signed_Zeros = True` és 0 eredmény esetén az előjel megegyezik `X` előjelével.

```
function S'Adjacent (X, Towards : T) return T;
```

Ha `Towards=X`, akkor a függvényérték `X`. Minden más esetben az `X`-szel `Towards` irányából közvetlenül szomszédos, a `T` típust ábrázoló gépi számot adja vissza, feltéve, hogy ilyen szám létezik. Ha az eredmény az `S` értékkészletén kívülre esne, `Constraint_Error` kivételt vált ki. Ha az eredmény 0 és `T'Signed_Zeros = True`, akkor az előjel megegyezik `X` előjelével.

```
function S'Copy_Sign (Value, Sign : T) return T;
```

Ha `Value` nem nulla, akkor azt a számot adja vissza, amelynek abszolút értéke megegyezik `Value` abszolút értékével, és előjele pedig megegyezik `Sign` előjelével. `Value = 0` esetén a visszatérési érték is nulla. Ha az eredmény az `S` értékkészletén kívülre esne, `Constraint_Error` kivételt válthat ki. Ha `S'Signed_Zeros = True`, akkor a 0 eredmény előjele megegyezik `Sign` előjelével.

```
function S'Leading_Part (X : T;
         Radix_Digits : universal_integer) return T;
```

Legyen `v := T'Machine_Radix`$^{\texttt{k-Radix_Digits}}$, ahol `k` az `X` paraméter normalizált kitevője. A függvényérték

- `Floor(X/v)*v`, ha `X` nem negatív és `Radix_Digits` pozitív,
- `Ceiling(X/v)*v`, ha `X` negatív és `Radix_Digits` pozitív.

Ha `Radix_Digits` nem pozitív, akkor `Constraint_Error` kivételt vált ki. Nulla eredmény esetén (ami csak `X=0` esetben lehetséges), a visszaadott érték előjele megegyezik `X` előjelével. Informálisan egy nem nulla `X` érték esetén az eredményt úgy kapjuk, hogy `X` számjegyeiből (a gépi számrendszerben értve) csak `Radix_Digits` értéket tartunk meg, a többit kinullázzuk.

```
function S'Machine (X : T) return T;
```

Ha `X` egy a `T`-t ábrázoló gépi szám, akkor a visszaadott érték `X`. Ellenkező esetben az `X` kerekítésével vagy vágásával kapható `X`-szel szomszédos valamely gépi számot adja. `Constraint_Error` kivételt vált ki, ha az `X` gépi számra kerekítése vagy vágása olyan gépi számot eredményez, amely kívül esik `S` értékkészletén. `S'Signed_Zeros = True` és nulla eredmény esetén a visszaadott érték előjele megegyezik `X` előjelével.

Egy tetszőleges `T` lebegőpontos típus bármely `S` altípusára értelmezve vannak még úgynevezett modell-orientált attribútumok, amelyeket a numerikus függelék tárgyalása után részletezünk.

A.4.4. A fixpontos típusok attribútumai

Egy tetszőleges `T` fixpontos típus bármely `S` altípusára a nyelv az alábbi attribútumokat definiálja:

```
S'Machine_Radix
```

A `T` típus gépi reprezentációja számrendszerének alapszáma. Az attribútum értéke `universal_integer` típusú.

```
S'Machine_Rounds
```

`True` értéket ad vissza, ha minden olyan előredefiniált művelet, amely `T` típusú értéket ad vissza nem pontos eredmény esetén kerekítést végez. Ellenkező esetben `False` értéket ad.

```
S'Machine_Overflows
```

`True` értéket ad, ha minden olyan előredefiniált művelet, amelynek értéke `T` típusú, a túlcsordulást és a nullával való osztást `Constraint_Error` kivétel kiváltásával jelzi. Egyébként a visszaadott érték `False`.

A.5. Input-Output

Az input-output műveleteket a nyelv előredefiniált csomagokon keresztül valósítja meg. Mindegyik csomag az `Ada` alapcsomag gyereke.

Ezek a csomagok a következők:

Sequential_IO és Direct_IO sablon csomagok olyan fájlokon alkalmazhatók, amelyek egy adott típusú elemekből tevődnek össze.

Storage_IO sablon csomag egy memória buffer írását, olvasását teszi lehetővé.

Text_IO és Wide_Text_IO csomagok szöveges fájlok kezeléséhez biztosítanak műveleteket.

Streams.Stream_IO és Text_IO.Text_Streams különböző típusú elemekből álló fájlok estén használatosak.

IO_Exceptions a fájlkezelés során fellépő kivételeket tartalmazza.

A.5.1. A `Sequential_IO` sablon csomag deklarációja

```
with Ada.IO_Exceptions;
generic
   type Element_Type(<>) is private;
package Ada.Sequential_IO is

   type File_Type is limited private;
   type File_Mode is (In_File, Out_File, Append_File);
```

Fájlkezelés:

```
   procedure Create(File : in out File_Type;
                    Mode : in File_Mode := Out_File;
                    Name : in String := "";
                    Form : in String := "");

   procedure Open  (File : in out File_Type;
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "");

   procedure Close (File : in out File_Type);
   procedure Delete(File : in out File_Type);
   procedure Reset (File : in out File_Type;
                    Mode : in File_Mode);
   procedure Reset (File : in out File_Type);

   function Mode  (File : in File_Type) return File_Mode;
   function Name  (File : in File_Type) return String;
   function Form  (File : in File_Type) return String;

   function Is_Open(File : in File_Type) return Boolean;
```

Input-output műveletek:

```
   procedure Read  (File : in File_Type;
                    Item : out Element_Type);
   procedure Write (File : in File_Type;
                    Item : in Element_Type);

   function End_Of_File(File : in File_Type) return Boolean;
```

Kivételek:

```
   Status_Error : exception
      renames IO_Exceptions.Status_Error;
   Mode_Error   : exception renames IO_Exceptions.Mode_Error;
   Name_Error   : exception renames IO_Exceptions.Name_Error;
   Use_Error    : exception renames IO_Exceptions.Use_Error;
   Device_Error : exception
      renames IO_Exceptions.Device_Error;
   End_Error    : exception renames IO_Exceptions.End_Error;
   Data_Error   : exception renames IO_Exceptions.Data_Error;

 private
```

```
   ...
end Ada.Sequential_IO;
```

A.5.2. A Direct_IO sablon csomag deklarációja

```
with Ada.IO_Exceptions;
generic
   type Element_Type is private;
package Ada.Direct_IO is

   type File_Type is limited private;
   type File_Mode is (In_File, Inout_File, Out_File);
   type Count is range 0 .. Implementation-Defined;
   subtype Positive_Count is Count range 1 .. Count'Last;
```

Fájlkezelés:

```
   procedure Create(File : in out File_Type;
                    Mode : in File_Mode := Inout_File;
                    Name : in String := "";
                    Form : in String := "");
   procedure Open  (File : in out File_Type;
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "");
   procedure Close (File : in out File_Type);
   procedure Delete(File : in out File_Type);
   procedure Reset (File : in out File_Type;
                    Mode : in File_Mode);
   procedure Reset (File : in out File_Type);

   function Mode  (File : in File_Type) return File_Mode;
   function Name  (File : in File_Type) return String;
   function Form  (File : in File_Type) return String;

   function Is_Open(File : in File_Type) return Boolean;
```

Input-output műveletek:

```
   procedure Read (File : in File_Type;
                   Item : out Element_Type;
                   From : in Positive_Count);
   procedure Read (File : in File_Type;
                   Item : out Element_Type);
```

```
   procedure Write(File : in File_Type;
                   Item : in Element_Type;
                   To   : in Positive_Count);
   procedure Write(File : in File_Type;
                   Item : in Element_Type);
   procedure Set_Index(File : in File_Type;
                       To : in Positive_Count);
   function Index(File : in File_Type) return Positive_Count;
   function Size (File : in File_Type) return Count;
   function End_Of_File(File : in File_Type) return Boolean;
```

Kivételek:

```
   Status_Error : exception renames
      IO_Exceptions.Status_Error;
   Mode_Error   : exception renames
      IO_Exceptions.Mode_Error;
   Name_Error   : exception renames
      IO_Exceptions.Name_Error;
   Use_Error    : exception renames
      IO_Exceptions.Use_Error;
   Device_Error : exception renames
      IO_Exceptions.Device_Error;
   End_Error    : exception renames
      IO_Exceptions.End_Error;
   Data_Error   : exception renames
      IO_Exceptions.Data_Error;

 private
    ...
end Ada.Direct_IO;
```

A.5.3. A Storage_IO sablon csomag deklarációja

```
with Ada.IO_Exceptions;
with System.Storage_Elements;
generic
   type Element_Type is private;
package Ada.Storage_IO is
   pragma Preelaborate(Storage_IO);
   Buffer_Size :
         constant System.Storage_Elements.Storage_Count
         := Implementation-Defined;
   subtype Buffer_Type is
```

```
    System.Storage_Elements.Storage_Array(1..Buffer_Size);
```

Input-output műveletek:

```
   procedure Read (Buffer : in Buffer_Type;
                   Item: out Element_Type);
   procedure Write(Buffer : out Buffer_Type;
                   Item : in Element_Type);
```

Kivételek:

```
   Data_Error  : exception renames IO_Exceptions.Data_Error;
end Ada.Storage_IO;
```

A.5.4. A Text_IO csomag deklarációja

```
with Ada.IO_Exceptions;
package Ada.Text_IO is

   type File_Type is limited private;
   type File_Mode is (In_File, Out_File, Append_File);
   type Count is range 0 .. Implementation-Defined;
   subtype Positive_Count is Count range 1 .. Count'Last;
   Unbounded : constant Count := 0;
```

Sor- és oldalhosszak:

```
   subtype Field is Integer range 0 .. Implementation-Defined;
   subtype Number_Base is Integer range 2 .. 16;
   type Type_Set is (Lower_Case, Upper_Case);
```

Fájlkezelés:

```
   procedure Create (File : in out File_Type;
                     Mode : in File_Mode := Out_File;
                     Name : in String  := "";
                     Form : in String  := "");
   procedure Open  (File : in out File_Type;
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "");
   procedure Close  (File : in out File_Type);
   procedure Delete (File : in out File_Type);
   procedure Reset  (File : in out File_Type;
                     Mode : in File_Mode);
   procedure Reset  (File : in out File_Type);
```

```
function Mode   (File : in File_Type) return File_Mode;
function Name   (File : in File_Type) return String;
function Form   (File : in File_Type) return String;
function Is_Open(File : in File_Type) return Boolean;
```

Alapértelmezett input ś output fájlok kezelśese:

```
procedure Set_Input (File : in File_Type);
procedure Set_Output(File : in File_Type);
procedure Set_Error (File : in File_Type);

function Standard_Input return File_Type;
function Standard_Output return File_Type;
function Standard_Error return File_Type;

function Current_Input return File_Type;
function Current_Output return File_Type;
function Current_Error return File_Type;

type File_Access is access constant File_Type;

function Standard_Input return File_Access;
function Standard_Output return File_Access;
function Standard_Error return File_Access;

function Current_Input return File_Access;
function Current_Output return File_Access;
function Current_Error return File_Access;
```

Buffer kezelése:

```
procedure Flush (File : in out File_Type);
procedure Flush;
```

Sor- és oldalhosszok kezelése:

```
procedure Set_Line_Length(File : in File_Type;
                          To : in Count);
procedure Set_Line_Length(To   : in Count);

procedure Set_Page_Length(File : in File_Type;
                          To : in Count);
procedure Set_Page_Length(To   : in Count);

function Line_Length(File : in File_Type) return Count;
```

```
function Line_Length return Count;

function Page_Length(File : in File_Type) return Count;
function Page_Length return Count;
```

Oszlopok, sorok és oldalak kezelése:

```
procedure New_Line  (File   : in File_Type;
                     Spacing : in Positive_Count := 1);
procedure New_Line  (Spacing : in Positive_Count := 1);

procedure Skip_Line  (File    : in File_Type;
                      Spacing : in Positive_Count := 1);
procedure Skip_Line  (Spacing : in Positive_Count := 1);

function End_Of_Line(File : in File_Type) return Boolean;
function End_Of_Line return Boolean;

procedure New_Page  (File : in File_Type);
procedure New_Page;

procedure Skip_Page  (File : in File_Type);
procedure Skip_Page;

function End_Of_Page(File : in File_Type) return Boolean;
function End_Of_Page return Boolean;

function End_Of_File(File : in File_Type) return Boolean;
function End_Of_File return Boolean;

procedure Set_Col (File : in File_Type;
                   To : in Positive_Count);
procedure Set_Col (To   : in Positive_Count);

procedure Set_Line(File : in File_Type;
                   To : in Positive_Count);
procedure Set_Line(To   : in Positive_Count);

function Col (File : in File_Type) return Positive_Count;
function Col return Positive_Count;

function Line(File : in File_Type) return Positive_Count;
function Line return Positive_Count;

function Page(File : in File_Type) return Positive_Count;
```

```
function Page return Positive_Count;
```

Karakterek inputja, outputja.

```
procedure Get(File : in File_Type; Item : out Character);
procedure Get(Item : out Character);

procedure Put(File : in File_Type; Item : in Character);
procedure Put(Item : in Character);

procedure Look_Ahead (File   : in File_Type;
                      Item   : out Character;
                      End_Of_Line : out Boolean);
procedure Look_Ahead (Item   : out Character;
                      End_Of_Line : out Boolean);

procedure Get_Immediate(File   : in File_Type;
                        Item   : out Character);
procedure Get_Immediate(Item   : out Character);
procedure Get_Immediate(File   : in File_Type;
                        Item   : out Character;
                        Available : out Boolean);
procedure Get_Immediate(Item   : out Character;
                        Available : out Boolean);
```

Sztringek inputja, outputja.

```
procedure Get(File : in File_Type; Item : out String);
procedure Get(Item : out String);

procedure Put(File : in File_Type; Item : in String);
procedure Put(Item : in String);

procedure Get_Line(File : in File_Type;
                   Item : out String;
                   Last : out Natural);
procedure Get_Line(Item : out String; Last : out Natural);
procedure Put_Line(File : in File_Type; Item : in String);
procedure Put_Line(Item : in String);
```

Egész típusok input-outputjához szükséges csomagok:

```
generic
   type Num is range <>;
package Integer_IO is
   Default_Width : Field := Num'Width;
```

```
      Default_Base  : Number_Base := 10;

      procedure Get(File  : in File_Type;
                    Item  : out Num;
                    Width : in Field := 0);
      procedure Get(Item  : out Num;
                    Width : in Field := 0);

      procedure Put(File  : in File_Type;
                    Item  : in Num;
                    Width : in Field := Default_Width;
                    Base  : in Number_Base := Default_Base);
      procedure Put(Item  : in Num;
                    Width : in Field := Default_Width;
                    Base  : in Number_Base := Default_Base);
      procedure Get(From : in String;
                    Item : out Num; Last : out Positive);
      procedure Put(To   : out String;
                    Item : in Num;
                    Base : in Number_Base := Default_Base);

   end Integer_IO;

   generic
      type Num is mod <>;
   package Modular_IO is
      Default_Width : Field := Num'Width;
      Default_Base  : Number_Base := 10;

      procedure Get(File  : in File_Type;
                    Item  : out Num;
                    Width : in Field := 0);
      procedure Get(Item  : out Num;
                    Width : in Field := 0);

      procedure Put(File  : in File_Type;
                    Item  : in Num;
                    Width : in Field := Default_Width;
                    Base  : in Number_Base := Default_Base);
      procedure Put(Item  : in Num;
                    Width : in Field := Default_Width;
                    Base  : in Number_Base := Default_Base);
      procedure Get(From : in String;
                    Item : out Num;
```

```
                     Last : out Positive);
      procedure Put(To   : out String;
                     Item : in Num;
                     Base : in Number_Base := Default_Base);
   end Modular_IO;
```

Valós típusok input-outputjához szükséges csomagok:

```
   generic
      type Num is digits <>;
   package Float_IO is
      Default_Fore : Field := 2;
      Default_Aft  : Field := Num'Digits-1;
      Default_Exp  : Field := 3;

      procedure Get(File  : in File_Type;
                     Item  : out Num;
                     Width : in Field := 0);
      procedure Get(Item  : out Num;
                     Width : in Field := 0);

      procedure Put(File : in File_Type;
                     Item : in Num;
                     Fore : in Field := Default_Fore;
                     Aft  : in Field := Default_Aft;
                     Exp  : in Field := Default_Exp);
      procedure Put(Item : in Num;
                     Fore : in Field := Default_Fore;
                     Aft  : in Field := Default_Aft;
                     Exp  : in Field := Default_Exp);

      procedure Get(From : in String;
                     Item : out Num;
                     Last : out Positive);
      procedure Put(To   : out String;
                     Item : in Num;
                     Aft  : in Field := Default_Aft;
                     Exp  : in Field := Default_Exp);
   end Float_IO;


   generic
      type Num is delta <>;
   package Fixed_IO is
```

```
      Default_Fore : Field := Num'Fore;
      Default_Aft  : Field := Num'Aft;
      Default_Exp  : Field := 0;

      procedure Get(File  : in File_Type;
                    Item  : out Num;
                    Width : in Field := 0);
      procedure Get(Item  : out Num;
                    Width : in Field := 0);

      procedure Put(File : in File_Type;
                    Item : in Num;
                    Fore : in Field := Default_Fore;
                    Aft  : in Field := Default_Aft;
                    Exp  : in Field := Default_Exp);
      procedure Put(Item : in Num;
                    Fore : in Field := Default_Fore;
                    Aft  : in Field := Default_Aft;
                    Exp  : in Field := Default_Exp);

      procedure Get(From : in String;
                    Item : out Num;
                    Last : out Positive);
      procedure Put(To   : out String;
                    Item : in Num;
                    Aft  : in Field := Default_Aft;
                    Exp  : in Field := Default_Exp);
   end Fixed_IO;

   generic
      type Num is delta <> digits <>;
   package Decimal_IO is
      Default_Fore : Field := Num'Fore;
      Default_Aft  : Field := Num'Aft;
      Default_Exp  : Field := 0;

      procedure Get(File  : in File_Type;
                    Item  : out Num;
                    Width : in Field := 0);
      procedure Get(Item  : out Num;
                    Width : in Field := 0);

      procedure Put(File : in File_Type;
                    Item : in Num;
```

```
                     Fore : in Field := Default_Fore;
                     Aft  : in Field := Default_Aft;
                     Exp  : in Field := Default_Exp);
      procedure Put(Item : in Num;
                     Fore : in Field := Default_Fore;
                     Aft  : in Field := Default_Aft;
                     Exp  : in Field := Default_Exp);

      procedure Get(From : in String;
                     Item : out Num; Last : out Positive);
      procedure Put(To   : out String;
                     Item : in Num;
                     Aft  : in Field := Default_Aft;
                     Exp  : in Field := Default_Exp);
   end Decimal_IO;
```

Felsorolási típus input-outputjához szükséges csomagok:

```
   generic
      type Enum is (<>);
   package Enumeration_IO is
      Default_Width   : Field := 0;
      Default_Setting : Type_Set := Upper_Case;

      procedure Get(File : in File_Type;
                     Item : out Enum);
      procedure Get(Item : out Enum);

      procedure Put(File  : in File_Type;
                     Item  : in Enum;
                     Width : in Field    := Default_Width;
                     Set   : in Type_Set := Default_Setting);
      procedure Put(Item  : in Enum;
                     Width : in Field    := Default_Width;
                     Set   : in Type_Set := Default_Setting);
      procedure Get(From : in String;
                     Item : out Enum;
                     Last : out Positive);
      procedure Put(To   : out String;
                     Item : in Enum;
                     Set  : in Type_Set := Default_Setting);
   end Enumeration_IO;
```

Kivételek:

```
   Status_Error : exception renames
```

```
      IO_Exceptions.Status_Error;
   Mode_Error   : exception renames
      IO_Exceptions.Mode_Error;
   Name_Error   : exception renames
      IO_Exceptions.Name_Error;
   Use_Error    : exception renames
      IO_Exceptions.Use_Error;
   Device_Error : exception renames
      IO_Exceptions.Device_Error;
   End_Error    : exception renames
      IO_Exceptions.End_Error;
   Data_Error   : exception renames
      IO_Exceptions.Data_Error;
   Layout_Error : exception renames
      IO_Exceptions.Layout_Error;

   private
      ...
end Ada.Text_IO;
```

A.5.5. A Streams.Stream_IO csomag deklarációja

```
with Ada.IO_Exceptions;
package Ada.Streams.Stream_IO is

   type Stream_Access is access all Root_Stream_Type'Class;
   type File_Type is limited private;
   type File_Mode is (In_File, Out_File, Append_File);
   type Count is range 0 .. Implementation-Defined;
   subtype Positive_Count is Count range 1 .. Count'Last;

   procedure Create (File : in out File_Type;
                     Mode : in File_Mode := Out_File;
                     Name : in String   := "";
                     Form : in String   := "");

   procedure Open (File : in out File_Type;
                   Mode : in File_Mode;
                   Name : in String;
                   Form : in String := "");

   procedure Close  (File : in out File_Type);
   procedure Delete (File : in out File_Type);
   procedure Reset  (File : in out File_Type;
```

```
                       Mode : in File_Mode);
   procedure Reset  (File : in out File_Type);

   function Mode (File : in File_Type) return File_Mode;
   function Name (File : in File_Type) return String;
   function Form (File : in File_Type) return String;

   function Is_Open  (File : in File_Type) return Boolean;
   function End_Of_File (File : in File_Type) return Boolean;
   function Stream (File : in File_Type) return Stream_Access;
```

Stream mutatót ad vissza, melyet a T'Input és a T'Output műveletek használnak.

```
   procedure Read (File : in File_Type;
                   Item : out Stream_Element_Array;
                   Last : out Stream_Element_Offset;
                   From : in Positive_Count);

   procedure Read (File : in File_Type;
                   Item : out Stream_Element_Array;
                   Last : out Stream_Element_Offset);

   procedure Write (File : in File_Type;
                    Item : in Stream_Element_Array;
                    To   : in Positive_Count);

   procedure Write (File : in File_Type;
                    Item : in Stream_Element_Array);

   procedure Set_Index(File : in File_Type;
                       To : in Positive_Count);
   function Index(File : in File_Type) return Positive_Count;
   function Size (File : in File_Type) return Count;

   procedure Set_Mode(File : in out File_Type;
                      Mode : in File_Mode);
   procedure Flush(File : in out File_Type);
```

Kivételek:

```
   Status_Error : exception
      renames IO_Exceptions.Status_Error;
   Mode_Error   : exception
      renames IO_Exceptions.Mode_Error;
```

```
   Name_Error  : exception
      renames IO_Exceptions.Name_Error;
   Use_Error  : exception
      renames IO_Exceptions.Use_Error;
   Device_Error : exception
      renames IO_Exceptions.Device_Error;
   End_Error  : exception
      renames IO_Exceptions.End_Error;
   Data_Error  : exception
      renames IO_Exceptions.Data_Error;

   private
      ...
end Ada.Streams.Stream_IO;
```

A.5.6. A Text_IO.Text_Streams csomag deklarációja

Text_IO.Text_Streams csomagokon keresztül a szövegfájlokat adatfolyam fájlokként kezelhetjük. A csomag deklarációja az alábbi:

```
with Ada.Streams;
package Ada.Text_IO.Text_Streams is
 type Stream_Access is
    access all Streams.Root_Stream_Type'Class;

 function Stream (File : in File_Type) return Stream_Access;
end Ada.Text_IO.Text_Streams;
```

A.5.7. A Wide_Text_IO.Text_Streams csomag

```
with Ada.Streams;
package Ada.Wide_Text_IO.Text_Streams is
 type Stream_Access is
    access all Streams.Root_Stream_Type'Class;

 function Stream (File : in File_Type) return Stream_Access;
end Ada.Wide_Text_IO.Text_Streams;
```

A fenti csomag Wide_Text változata ugyanarra a célra használható, csak Wide_Text_IO-beli fájlokra.

A.5.8. Az IO_Exceptions csomag deklarációja

```
package Ada.IO_Exceptions is
 pragma Pure(IO_Exceptions);

 Status_Error : exception;
 Mode_Error : exception;
 Name_Error : exception;
 Use_Error  : exception;
 Device_Error : exception;
 End_Error  : exception;
 Data_Error : exception;
 Layout_Error : exception;

end Ada.IO_Exceptions;
```

A.6. A Command_Line csomag

A `Command_Line` csomagon keresztül a program el tudja érni az indításkor kapott argumentumait, illetve be tudja állítani a normális befejeződéskor visszaadandó kilépési kódot. A csomag definíciója a következő:

```
package Ada.Command_Line is
  pragma Preelaborate(Command_Line);

  function Argument_Count return Natural;
```

Ha a külső környezet támogatja argumentumok átadását a programnak, akkor a függvény visszaadja a programnak átadott argumentumok számát. Ha a környezet az ilyen paraméterátadást nem támogatja, akkor a függvényérték 0. Az „argumentumok száma” kifejezés értelmezése implementációnként változhat.

```
  function Argument (Number : in Positive) return String;
```

Ha a külső környezet támogatja argumentumok átadását a programnak, akkor az `Argument` függvény visszaadja a `Number` relatív pozíción található argumentumot (implementáció által definiált formában). Ha `Number` nem esik bele az `1..Argument_Count` intervallumba, akkor `Constraint_Error` hibajelzést kapunk. Ha a környezet nem támogatja a parancssor argumentumokat, akkor mindig `Constraint_Error` hibajelzést kapunk.

```
  function Command_Name return String;
```

Ha a környezet támogatja a parancssor argumentumok használatát, akkor a `Command_Name` függvény visszaadja a programot indító parancsot az implementáció által definiált formában, ellenkező esetben a visszaadott érték null sztring.

```
   type Exit_Status is implementation-defined integer type;

   Success : constant Exit_Status;
   Failure : constant Exit_Status;

   procedure Set_Exit_Status (Code : in Exit_Status);
private
   ...
end Ada.Command_Line;
```

Az `Exit_Status` típus a külső környezetben megengedett visszatérési értékek értékhalmazát reprezentálja. A `Success` konstans a sikeres, a `Faliure` konstans a sikertelen végrehajtás visszatérési értékeit reprezentálja. Ha a külső környezet támogatja a programok visszatérési értékének kezelését, akkor a visszatérési kódot a `Set_Exit_Status` művelettel állíthatjuk be. Sikeres végrehajtás esetén a program visszatérési értéke a legutolsó `Set_Exit_Status` művelettel beállított érték (ennek hiányában a `Success` konstans) lesz. Abnormális befejeződésre a `Set_Exit_Status` műveletnek nincs hatása. Ha a külső környezet nem támogatja a programok visszatérési értékének a kezelését, akkor az eljárásnak nincs hatása.

B. Függelék

Kapcsolódás más nyelvekhez

Ebben a függelékben megvizsgáljuk, hogy az Ada milyen eszközökkel támogatja a vegyes nyelven írt programokat. Először az általános fogalmakat tekintjük át, majd külön megnézzük a C, COBOL és Fortran nyelvekhez való kapcsolódást támogató csomagokat.

B.1. A kapcsolódást leíró pragmák

Az `Import` pragmát használhatjuk arra, hogy egy más nyelven megírt programegységet egy Ada programban elérhessünk. Így például ezen keresztül lehetővé tehetjük egy más nyelven megírt eljárás Adából történő meghívását, vagy egy más nyelven deklarált változó Ada programbeli használatát.

Egy Ada programegységet az `Export` pragmán keresztül tehetünk más nyelven írt programrészek számára elérhetővé.

A fenti két pragma szintaxisa az alábbi:

```
pragma Import(
    [Convention =>] konvenció-azonosító,
    [Entity =>] lokális-név
    [, [External_Name =>] sztring-kifejezés]
    [, [Link_Name =>] sztring-kifejezés]);

pragma Export(
    [Convention =>] konvenció-azonosító,
    [Entity =>] lokális-név
    [, [External_Name =>] sztring-kifejezés]
    [, [Link_Name =>] sztring-kifejezés]);
```

A konvenció-azonosító a konvenció neve. Az érvényes konvenciónevek halmaza implementációnként változhat. A lokális-név a programelem Adabeli neve. Az `External_Name` mezőben sztring formában az objektum külső

nyelvbeli nevét, míg a `Link_Name` mezőben az objektum linker által használt nevét adhatjuk meg. Ez utóbbi két mező opcionális.

Az `Import` pragma a deklaráció végrehajtásának felel meg. Ezért aztán ugyanabban a deklarációs részben, csomag specifikációban, taszk definíciós részben vagy védett definícióban kell előfordulnia, mint a hivatkozott objektum deklarációja. Ha a lokális név több Ada programegységet is azonosít, akkor az `Import` pragma mindegyikük deklarációjának végrehajtását jelenti.

Egy importált objektum deklarációjában nem szerepelhet explicit inicializáló kifejezés.

A `Convention` pragmával megadhatjuk, hogy egy Ada programegységnek egy más nyelv konvencióit kell használnia. Ezt általában típusokra és callback-eljárásokra használjuk.

Legyen L egy konvenció-azonosító egy külső nyelvhez. Ekkor azt mondjuk, hogy a `T` típus L-kompatíbilis, ha az alábbi feltételek valamelyike teljesül:

- A `T` típus egy az L konvencióhoz kapcsolódó interfész csomagban lett deklarálva, és ott definíció szerint L-kompatíbilis.
- A `Convention` pragmán keresztül a `T` típushoz hozzá lett rendelve az L konvenció, és a `T` típusra alkalmazható az L konvenció, azaz:
 - `T` egy olyan tömb típus, amelynek elemtípusa L-kompatíbilis,
 - `T` egy olyan rekord típus, amelynek nincs diszkriminánsa, és minden komponense L-kompatíbilis,
 - `T` egy L-kompatíbilis típusra mutató pointer típus,
 - `T` olyan eljárásra mutató pointer típus, amelynek mind paraméterei, mind visszatérési értéke L-kompatíbilisek.
- `T` egy L-kompatíbilis típusból lett származtatva.
- A nyelv implementációja T-t L-kompatíbilisnek definiálja.

Az importált és exportált objektumok típusának kompatíbilisnek kell lennie a megfelelő `Import` és `Export` pragmákban megadott konvencióval.

A `Linker_Options` pragmán keresztül az Ada programban a linkernek szóló paramétereket helyezhetünk el. Az itt megadható sztring szintaxisa, illetve az egyéb rá vonatkozó szabályok a nyelv implementációjától függenek.

A fenti pragmák használatát az alábbi kis csomagdefiníción keresztül szemléltetjük:

```
package Fortran_Library is
  function Sqrt (X : Float) return Float;
  function Exp  (X : Float) return Float;
private
```

```
    pragma Import(Fortran, Sqrt);
    pragma Import(Fortran, Exp);
  end Fortran_Library;
```

B.2. Az Interfaces csomag

Az `Interfaces` csomag számos olyan csomagnak a szülője, amelyek a más nyelveken megírt programrészekkel való kapcsolattartást támogatják. Tartalmaz néhány olyan – a nyelv implementációjától függő – típusdeklarációt, amely több nyelv felé is használható lehet. A csomag vázlatos felépítése:

```
package Interfaces is
  pragma Pure(Interfaces);

  type Integer_n is range -2**(n-1) .. 2**(n-1) - 1;

  type Unsigned_n is mod 2**n;

  function Shift_Left(Value: Unsigned_n; Amount: Natural)
     return Unsigned_n;
  function Shift_Right(Value: Unsigned_n; Amount: Natural)
     return Unsigned_n;
  function Shift_Right_Arithmetic (Value: Unsigned_n;
                                   Amount: Natural)
     return Unsigned_n;
  function Rotate_Left(Value: Unsigned_n; Amount: Natural)
     return Unsigned_n;
  function Rotate_Right(Value : Unsigned_n; Amount: Natural)
     return Unsigned_n;
  ...
end Interfaces;
```

A nyelv minden implementációjában az `Interfaces` csomag látható részében kell szerepelniük az alábbi deklarációknak:

- Az `n` bites előjeles és előjel nélküli egészek típusa, minden olyan `n`-re, amelyet az adott környezet támogat. A kettes komplemensben ábrázolt előjeles egész típusok neve `Integer_n`, míg az előjel nélkülieké `Unsigned_n`.
- A csomagban deklarált összes előjel nélküli egész típusra a balra és jobbra történő bitenkénti eltolás és ciklikus eltolás műveletek. A műveletekben szereplő `Amount` paraméterben a mozgatás méretét adhatjuk meg. Az eltolás műveletekben a behozott bitek nullák, kivéve

a `Shift_Right_Arithmetic` művelet, ahol egyesek jönnek be, ha a `Value` paraméter legfelső bitje 1.

- A hardware által támogatott lebegőpontos típusok deklarációi. Ezek nevére a nyelvdefiníció nem tartalmaz megkötéseket.

Minden, a nyelv implementációja által támogatott konvenció-azonosítóhoz általában található egy az `Interfaces` csomagból származó gyerek csomag. Ezekben a gyerek csomagokban találhatók azok a deklarációk, amelyek hasznosak lehetnek az adott nyelvhez való kapcsolódásban.

B.3. Kapcsolódás a C nyelvhez

A C nyelvhez való kapcsolódást támogató típusok, eljárások és függvények az `Interfaces.C` csomagban és gyerekeiben találhatók. Ezek a csomagok az `Import`, `Export` és `Convention` pragmákat a `C` konvenció-azonosítóval támogatják.

Az `Interfaces.C` csomag azokat az alapvető típusokat, konstansokat és műveleteket tartalmazza, amelyeken keresztül egy Ada program skaláris és sztring értékeket tud átadni C függvényeknek:

```
package Interfaces.C is
   pragma Pure(C);

   CHAR_BIT  : constant := implementation-defined;
      -- typically 8
   SCHAR_MIN : constant := implementation-defined;
      -- typically -128
   SCHAR_MAX : constant := implementation-defined;
      -- typically 127
   UCHAR_MAX : constant := implementation-defined;
      -- typically 255
```

A fenti deklarációk a C nyelv `<limits.h>` header fájljának definícióin alpulnak.

```
   type int   is range implementation-defined;
   type short is range implementation-defined;
   type long  is range implementation-defined;

   type signed_char is range SCHAR_MIN .. SCHAR_MAX;
   for signed_char'Size use CHAR_BIT;

   type unsigned is mod implementation-defined;
```

```
type unsigned_short is mod implementation-defined;
type unsigned_long  is mod implementation-defined;

type unsigned_char is mod (UCHAR_MAX+1);
for unsigned_char'Size use CHAR_BIT;

subtype plain_char is implementation-defined;

type ptrdiff_t is range implementation-defined;

type size_t is mod implementation-defined;

type C_float     is digits implementation-defined;

type double      is digits implementation-defined;

type long_double is digits implementation-defined;

type char is <implementation-defined character type>;

nul : constant char := char'First;
```

A csomagban deklarált összes típus C-kompatíbilis. Az `int`, `short`, `long`, `unsigned`, `ptrdiff_t`, `size_t`, `double`, `char` és `wchar_t` típusok az azonos nevű C típushoz illeszkednek. Hasonlóan a `signed_char`, `unsigned_short`, `unsigned_long`, `unsigned_char`, `C_float`, és `long_double` típusok pedig a `signed char`, `unsigned short`, `unsigned long`, `unsigned char`, `float`, and `long double` típusoknak felelnek meg.

```
function To_C   (Item : in Character) return char;

function To_Ada (Item : in char) return Character;
```

A `To_C` és `To_Ada` függvények az Ada `Character` és a C `char` típusa közötti leképezést valósítják meg.

```
type char_array is array (size_t range <>) of aliased char;
pragma Pack(char_array);
for char_array'Component_Size use CHAR_BIT;
```

A `char_array` típus a C karakter tömb típusának felel meg, de természetesen minden olyan C függvény hívásában is használható, amely `char*` paramétert vár. (Lásd még az `Interfaces.C.Strings` csomagot.)

```
function Is_Nul_Terminated (Item : in char_array)
   return Boolean;
```

True-t ad vissza, ha az Item tömb tartalmazza a nul karakter konstanst, különben a visszatérési érték False.

```
function To_C   (Item       : in String;
                 Append_Nul : in Boolean := True)
   return char_array;

function To_Ada (Item     : in char_array;
                 Trim_Nul : in Boolean := True)
   return String;
```

A To_C függvény értéke Append_Nul értékétől függően egy Item'Length (ha Append_Nul értéke False), vagy egy Item'Length+1 hosszú tömb, melynek legkisebb indexe 0. A visszaadott vektor elemeit úgy kapjuk, hogy az Item paraméter megfelelő elemére alkalmazzuk a karakteres To_C függvényt. Ha Append_Nul értéke True, akkor az eredmény végére odakerül a nul konstans is.

A To_Ada függvény eredménye a Trim_Nul paraméter False értéke esetén egy Item'Length hosszúságú String. Trim_Nul = True esetén a visszaadott String hossza az Item paraméter első nul karakterét megelőző kezdőszelet hossza lesz. A visszaadott String kezdőindexe 1, tartalmát pedig úgy kapjuk meg, hogy a fent meghatározott hosszon az Item paraméter megfelelő karakterére alkalmazzuk a karakteres To_Ada függvényt. A függvény Terminator_Error kivételt vált ki, ha Trim_Nul értéke True és az Item paraméter nem tartalmazza a nul char értéket.

```
procedure To_C (Item       : in String;
                Target     : out char_array;
                Count      : out size_t;
                Append_Nul : in Boolean := True);

procedure To_Ada (Item     : in char_array;
                  Target   : out String;
                  Count    : out Natural;
                  Trim_Nul : in Boolean := True);
```

A To_C eljárás az Item sztring minden elemét a karakteres To_C függvényen keresztül átkonvertálja char típusú értékké, és elhelyezi a Target paraméter megfelelő pozícióján. Append_Nul = True esetén a Target paraméter utolsó feltöltött eleme mögé berak még egy nul értéket. Count értéke mindkét esetben a Target-ben feltöltött elemek számát (az esetleges nul értéket is beleszámítva) adja vissza. Ha a Target paraméter nem elég hosszú, az eljárás Constraint_Error kivételt vált ki.

A `To_Ada` eljárás `Item` minden elemét (`Trim_Nul = True` esetén az első `nul` értéket megelőző minden elemét) a karakteres `To_Ada` függvényen keresztül `Character` értékké konvertálja, és a `Target` sztring megfelelő elemének értékül adja. `Count` a `Target`-ben értékül adott elemek száma lesz. Ha a `Target` paraméter nem elég hosszú, akkor az eljárás `Constraint_Error` kivételt vált ki. Ha `Trim_Nul = True`, és `Item` nem tartalmaz `nul` értéket, akkor viszont `Terminator_Error` hibajelzést kapunk.

```
type wchar_t is implementation-defined;

wide_nul : constant wchar_t := wchar_t'First;

function To_C   (Item : in Wide_Character) return wchar_t;
function To_Ada (Item : in wchar_t)
   return Wide_Character;

type wchar_array is array (size_t range <>) of
   aliased wchar_t;

pragma Pack(wchar_array);

function Is_Nul_Terminated (Item : in wchar_array)
   return Boolean;
```

A `Wide_Character` és `Wide_String` típusok – valamint C megfelelőik – kezelése analóg a karakterek és sztringek kezelésével. Az `Is_Nul_Terminated` függvény visszaadja, hogy az `Item` paraméter tartalmazza-e a `wide_nul` értéket. A `To_C` és `To_Ada` függvények pedig megvalósítják a C `wchar` és az Ada `Wide_Character` típusai közötti leképezéseket.

```
function To_C   (Item       : in Wide_String;
                 Append_Nul : in Boolean := True)
   return wchar_array;

function To_Ada (Item     : in wchar_array;
                 Trim_Nul : in Boolean := True)
   return Wide_String;

procedure To_C (Item       : in  Wide_String;
                Target     : out wchar_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);
procedure To_Ada (Item     : in  wchar_array;
                  Target   : out Wide_String;
```

```
                        Count    : out Natural;
                        Trim_Nul : in  Boolean := True);
      Terminator_Error : exception;

   end Interfaces.C;
```

A To_C és To_Ada függvények és eljárások a String típusúaknál leírtak szerint működnek, értelemszerűen a nul érték helyett wide_nul érték használatával.

B.3.1. Az Interfaces.C.Strings csomag

Az Interfaces.C.Strings csomag azokat a típusokat és műveleteket deklarálja, amelyek segítségével C típusú sztringeket tudunk kezelni egy Ada programból. A csomag deklarációja:

```
package Interfaces.C.Strings is
   pragma Preelaborate(Strings);

   type char_array_access is access all char_array;

   type chars_ptr is private;

   type chars_ptr_array is array (size_t range <>)
      of chars_ptr;

   Null_Ptr : constant chars_ptr;
```

A chars_ptr (C-kompatíbilis) típus megfelel a C char* típusának. Egy chars_ptr objektum – hacsak nem importált – kezdeti értéke Null_Ptr. A char_array_access típus nem C-kompatíbilis, ezért az ilyen objektumokat chars_ptr objektumokká kell konvertálni a C-nek történő átadás előtt.

```
   function To_Chars_Ptr (Item      : in char_array_access;
                          Nul_Check : in Boolean := False)
      return chars_ptr;
```

Ha Item null, akkor a visszaadott érték Null_Ptr. Ellenkező esetben a függvény egy allokáció nélküli pointer konverziót hajt végre. Ha Item.all nem tartalmaz nul értéket és Nul_Check = True, akkor a függvényhívás Terminator_Error hibajelzést ad.

```
   function New_Char_Array (Chars : in char_array)
      return chars_ptr;
```

A függvény allokál egy a `Chars` paraméternek megfelelő méretű objektumot, feltölti a `Chars(Chars'First .. Index)` elemeket `nul` értékkel, és visszaad egy, az allokált objektumra hivatkozó mutatót. A fenti kifejezésben `Index` értéke `Chars'Last`, ha `Chars`-ban nem szerepel a `nul` érték, míg ellenkező esetben az a legkisebb `I size_t` érték, amire `Chars(I+1) = nul`. Ha a függvény nem tud az objektumnak memóriát foglalni, akkor `Storage_Error` kivételt vált ki.

```
function New_String (Str : in String) return chars_ptr;
```

Ez a függvény ekvivalens a `New_Char_Array(To_C(Str))` hívással.

```
procedure Free (Item : in out chars_ptr);
```

Ha `Item` nem `Null_Ptr`, akkor az eljárás felszabadítja a `Value(Item)` által elfoglalt memóriát és `Item`-et `Null_Ptr`-re állítja. Ellenkező esetben nincs hatása. Ha `Item` nem null-terminált sztringre mutat, vagy `Item` nem a `New_Char_Array` vagy `New_String` műveletek valamelyikén keresztül lett allokálva, akkor végrehajtása hibás.

```
Dereference_Error : exception;

function Value (Item : in chars_ptr) return char_array;
```

Ha `Item = Null_Ptr`, akkor `Dereference_Error` hibajelzést kapunk. Ellenkező esetben az `Item` által mutatott karaktertömb első `nul` értékéig tartó kezdőszeletét adja vissza (a `nul` értékkel együtt). Az eredmény kezdőindexe 0. Ha `Item` nem null-terminált sztringre mutat, akkor végrehajtása hibás.

```
function Value (Item : in chars_ptr; Length : in size_t)
   return char_array;
```

Ha `Item = Null_Ptr`, akkor `Dereference_Error` hibajelzést kapunk. Ellenkező esetben az alábbi két tömb közül a rövidebbet kapjuk vissza: Az `Item` által mutatott karaktertömb első `Length` eleme (nem feltétlenül null-terminált) vagy `Value(Item)`. A visszaadott érték kezdőindexe 0.

```
function Value (Item : in chars_ptr) return String;
```

Ekvivalens a `To_Ada(Value(Item), Trim_Nul=>True)` hívással.

```
function Value (Item : in chars_ptr; Length : in size_t)
   return String;
```

Ekvivalens a `To_Ada(Value(Item, Length), Trim_Nul=>True)` hívással.

```
   function Strlen (Item : in chars_ptr) return size_t;
```

Legyen `Val = Value(Item)`. Ekkor a visszaadott érték `Val'Length-1`. Ha `Item` `Null_Ptr`, akkor `Dereference_Error` hibajelzést kapunk. Végrehajtása hibás, ha az `Item` által mutatott karaktertömb nem tartalmaz `nul` értéket.

```
   Update_Error : exception;

   procedure Update (Item   : in chars_ptr;
                     Offset : in size_t;
                     Chars  : in char_array;
                     Check  : in Boolean := True);
```

Ez az eljárás belemásolja a `Chars` paraméter elemeit az `Item` által mutatott karaktertömbbe, az `Offset` pozíciótól kezdődően. Az `Item` által mutatott karaktertömb null terminátorára a `Check` paraméter `True`-ra állításával „vigyázhatunk": Legyen `N  = Strlen(Item)`. Ekkor ha `Check = True` akkor `Offset+Chars'Length>N` esetén `Update_Error` hibajelzést kapunk.

```
   procedure Update (Item   : in chars_ptr;
                     Offset : in size_t;
                     Str    : in String;
                     Check  : in Boolean := True);
```

Ekvivalens az `Update(Item, Offset, To_C(Str), Check)` hívással.

```
  private
     ... -- not specified by the language
  end Interfaces.C.Strings;
```

B.3.2. Az `Interfaces.C.Pointers` sablon csomag

Ez a csomag elérhetővé teszi az Ada programozó számára a C típusú pointerműveleteket. Definiálja a Pointer mutató típust a műveleteivel együtt. A C-hez hasonlóan a Pointer típus egy objektumát egy tömb első elemére mutató pointernek tekinti.

A csomag kétféle felhasználást tesz lehetővé: az egyikben a mutatott tömb egy speciális záróelemmel van terminálva, míg a másik megközelítés szerint a programozónak kell a tömb végére figyelnie. A csomag deklarációja a következő:

```
generic
   type Index is (<>);
   type Element is private;
```

```
   type Element_Array is array (Index range <>) of
      aliased Element;
   Default_Terminator : Element;
package Interfaces.C.Pointers is
   pragma Preelaborate(Pointers);

   type Pointer is access all Element;
```

A `Pointer` típus C-kompatíbilis és a C nyelv `Element*` pointer típusára illeszkedik. Egy ilyen típusú objektumot úgy kell értelmezni, mint egy `Element_Array` első elemére mutató pointert.

```
   function Value(Ref        : in Pointer;
                  Terminator : in Element
                     := Default_Terminator)
      return Element_Array;
```

Visszaadja a `Ref` paraméter által mutatott tömb azon kezdőszeletét, amely az első `Terminator` elemig tart (azzal bezárólag). Az eredmény tömb kezdőindexe `Index'First`. `Interfaces.C.Strings.Dereference_Error` kivételt vált ki, ha `Ref` értéke `null`.

```
   function Value(Ref    : in Pointer;
                  Length : in ptrdiff_t)
      return Element_Array;
```

A függvény visszaadja a `Ref` által mutatott tömb első `Length` eleméből képzett `Element_Array` objektumot. Ha a `Ref` paraméter értéke `null`, akkor a függvény `Interfaces.C.Strings.Dereference_Error` kivételt vált ki.

```
    Pointer_Error : exception;

   function "+" (Left : in Pointer;   Right : in ptrdiff_t)
      return Pointer;
   function "+" (Left : in ptrdiff_t; Right : in Pointer)
      return Pointer;
   function "-" (Left : in Pointer;   Right : in ptrdiff_t)
      return Pointer;
   function "-" (Left : in Pointer;   Right : in Pointer)
      return ptrdiff_t;
```

Ezek az infix operátorok a C típusú pointer-aritmetikát valósítják meg. Mindegyik függvény `Pointer_Error` kivételt vált ki, ha valamely `Pointer` típusú paramétere `null`.

```
procedure Increment (Ref : in out Pointer);
```

Ekvivalens a `Ref := Ref+1` értékadással.

```
procedure Decrement (Ref : in out Pointer);
pragma Convention (Intrinsic, "+");
pragma Convention (Intrinsic, "-");
pragma Convention (Intrinsic, Increment);
pragma Convention (Intrinsic, Decrement);
```

A `Decrement` függvény ekvivalens a `Ref := Ref-1` értékadással.

```
function Virtual_Length (Ref        : in Pointer;
                         Terminator : in Element
                            := Default_Terminator)
   return ptrdiff_t;
```

Visszaadja az első terminátor elemet megelőző elemek számát.

```
procedure Copy_Terminated_Array (Source     : in Pointer;
                                 Target     : in Pointer;
                                 Limit      : in ptrdiff_t
                                  := ptrdiff_t'Last;
                                 Terminator : in Element
                                  :=  Default_Terminator);
```

Ez az eljárás `Value(Source, Terminator)` értékét `Target`-be másolja. Ha a másolandó elemek száma nagyobb, mint `Limit`, akkor csak `Limit` számú elemet másol. Akár `Source`, akár `Target` null, `Dereference_Error` hibajelzést kapunk. Arra a programozónak kell figyelnie, hogy a céltömb hosszán ne másoljon túl.

```
procedure Copy_Array (Source  : in Pointer;
                      Target  : in Pointer;
                      Length  : in ptrdiff_t);

end Interfaces.C.Pointers;
```

Ez az eljárás a `Source` által mutatott tömb első `Length` elemét átmásolja a `Target` által mutatott tömbbe. `Dereference_Error` hibajelzést kapunk, ha `Source` vagy `Target` értéke `null`.

B.4. Kapcsolódás a COBOL nyelvhez

A COBOL nyelvhez történő kapcsolódás eszközeit az `Interfaces.COBOL` csomag tartalmazza. Ezen kívül a nyelv támogatja az `Import`, `Export` és `Convention` pragmák `COBOL` konvenció-azonosítóval történő használatát.

Az `Interfaces.COBOL` csomag deklarációja a következő:

```
package Interfaces.COBOL is
   pragma Preelaborate(COBOL);

   type Floating      is digits implementation-defined;
   type Long_Floating is digits implementation-defined;

   type Binary      is range implementation-defined;
   type Long_Binary is range implementation-defined;
```

A csomagban található összes típus COBOL-kompatíbilis. A `Floating` és `Long_Floating` típusok a COBOL nyelv azon natív típusainak felelnek meg, amelyek megvalósításai lebegőpontosak. Hasonlóan a `Binary` és `Long_Binary` típusok a Cobol nyelv bináris megvalósítású natív típusainak felelnek meg.

```
   Max_Digits_Binary: constant := implementation-defined;
   Max_Digits_Long_Binary: constant := implementation-defined;
```

A `Max_Digits_Binary` konstans értéke a `Binary` típusba eső numerikus értékek számjegyeinek maximális száma. Ugyanezt az jellemzőt tartalmazza a `Max_Digits_Long_Binary` konstans a `Long_Binary` típusra.

```
   type Decimal_Element  is mod implementation-defined;
   type Packed_Decimal is array (Positive range <>)
      of Decimal_Element;
   pragma Pack(Packed_Decimal);
```

A `Packed_Decimal` típus, a COBOL pakolt-decimális ábrázolásának felel meg.

```
   type COBOL_Character is
      implementation-defined character type;

   Ada_To_COBOL : array (Character) of COBOL_Character :=
      implementation-defined;

   COBOL_To_Ada : array (COBOL_Character) of Character :=
      implementation-defined;
```

A `COBOL_Character` típus a COBOL implementációban használt karaktertípus értékkészletét definiálja. Az `Ada_To_COBOL` és `COBOL_To_Ada` vektorok pedig a két nyelv karakterkészlete közötti leképezéseket definiálják.

```
type Alphanumeric is array (Positive range <>)
   of COBOL_Character;
pragma Pack(Alphanumeric);
```

Az `Alphanumeric` típus az COBOL alfanumerikus adatkategóriájának felel meg.

```
function To_COBOL (Item : in String) return Alphanumeric;
function To_Ada   (Item : in Alphanumeric) return String;
procedure To_COBOL (Item   : in String;
                    Target : out Alphanumeric;
                      Last : out Natural);

procedure To_Ada (Item     : in Alphanumeric;
                  Target   : out String;
                  Last     : out Natural);
```

A `To_COBOL` és `To_Ada` függvények az előbb bemutatott karakteres leképezés tömbök alapján a kapott sztringet konvertálják át. A visszaadott értékek hossza megegyezik a megadott paraméter hosszával, a kezdőindex pedig 1. Az ugyanilyen nevű eljárások a konverzió eredményét a megadott `Target` paraméterben helyezik el, a `Last` paraméterben visszaadva az utolsó kitöltött elem indexét (vagy nullát, ha `Item` null tömb). `Constraint_Error` hibajelzést kapunk, ha `Item'Length > Target'Length`.

```
type Numeric is array (Positive range <>)
   of COBOL_Character;
pragma Pack(Numeric);
```

A `Numeric` típus a COBOL megjelenítésre használt numerikus adatkategóriájának felel meg.

```
type Display_Format is private;

Unsigned             : constant Display_Format;
Leading_Separate     : constant Display_Format;
Trailing_Separate    : constant Display_Format;
Leading_Nonseparate  : constant Display_Format;
Trailing_Nonseparate : constant Display_Format;
```

```
type Binary_Format is private;

High_Order_First  : constant Binary_Format;
Low_Order_First   : constant Binary_Format;
Native_Binary     : constant Binary_Format;

type Packed_Format is private;

Packed_Unsigned   : constant Packed_Format;
Packed_Signed     : constant Packed_Format;
```

A `Display_Format`, `Binary_Format`, és `Packed_Format` típusokat az Ada decimális típus és a COBOL belső vagy külső adatreprezentációk közötti konverziókban használjuk. A `Native_Binary` konstans értéke az implementációtól függően `High_Order_First` vagy `Low_Order_First` értékével egyezik meg.

```
type Byte is mod 2**COBOL_Character'Size;
type Byte_Array is array (Positive range <>) of Byte;
pragma Pack (Byte_Array);
```

Adából a `Byte` és `Byte_Array` típusokon keresztül érhetők el a COBOL bináris adatai. Az Ada decimális típusai és a COBOL belső vagy külső reprezentációi közötti konverziót a `Decimal_Conversions` beágyazott sablon csomag valósítja meg (a sablon paramétere a numerikus típus):

```
Conversion_Error : exception;

generic
   type Num is delta <> digits <>;
package Decimal_Conversions is

   function Valid (Item   : in Numeric;
                   Format : in Display_Format)
      return Boolean;
```

A `Valid` függvény ellenőrzi, hogy az `Item` paraméter értéke illeszkedik-e a `Format` paraméterben megadott formátumra. A `Format` értékétől függően a visszatérési érték az alábbi lesz:

- `Format=Unsigned` esetén: Ha `Item` nulla vagy több bevezető szóköz karakterből és utána egy vagy több számjegyből áll, akkor a függvényérték `True`, különben `False`.

- `Format=Leading_Separate` esetén: Ha `Item` nulla vagy több bevezető szóköz karakterből, egy előjelből és utána egy vagy több számjegyből áll, akkor a függvényérték `True`, különben `False`.
- `Format=Trailing_Separate` esetén: Ha `Item` nulla vagy több bevezető szóköz karakterből és utána egy vagy több számjegyből, majd egy azt követő előjelből áll, akkor a függvényérték `True`, különben `False`.

Más `Format` értékek esetén a függvényhívás eredménye implementációfüggő.

```
function Length (Format : in Display_Format)
   return Natural;
```

A `Length` függvény visszaadja a `Num` típus minden értékének `Format` szerinti reprezentálásához szükséges `Numeric` érték minimális hosszát.

```
function To_Decimal (Item   : in Numeric;
                     Format : in Display_Format)
   return Num;
```

Az `Item` paraméter `Format` által értelmezett értékét adja vissza `Num` típusú értékként. A tizedesjegyek számát `Num'Scale` határozza meg, ezért nincs sem kerekítés sem jegylevágás. A függvényhívás `Conversion_Error` kivételt vált ki, ha az `Item` által reprezentált szám kívül esik a `Num` típus értékkészletén.

```
function To_Display (Item   : in Num;
                     Format : in Display_Format)
   return Numeric;
```

Ez a függvény az `Item` paraméterben kapott számot a `Format` paraméternek megfelelően `Numeric` értékké konvertálja. `Conversion_Error` hibajelzést kapunk, ha `Item` negatív, `Format` pedig `Unsigned`.

```
function Valid (Item   : in Packed_Decimal;
                Format : in Packed_Format) return
   Boolean;
```

A függvény akkor ad `True` értéket, ha az `Item` paraméter értéke illeszkedik a `Format` paraméterhez. Ellenkező esetben a függvényérték `False`. A `Packed_Decimal` értékek képzési szabályai implementációfüggőek.

```
function Length (Format : in Packed_Format)
   return Natural;
```

A `Length` függvény visszaadja a `Num` típus minden értékének `Format` szerinti reprezentálásához szükséges `Packed_Decimal` érték minimális hosszát.

```
function To_Decimal (Item   : in Packed_Decimal;
                     Format : in Packed_Format)
   return Num;
```

Egy `Num` típusú értéket állít elő az `Item` paraméter `Format` szerinti értelmezése alapján. A tizedesjegyek számát `Num'Scale` határozza meg. A függvényhívás `Conversion_Error` kivételt vált ki, ha az `Item` által reprezentált szám kívül esik a `Num` típus értékkészletén.

```
function To_Packed (Item   : in Num;
                    Format : in Packed_Format)
   return Packed_Decimal;
```

A függvény a kapott szám `Format` alapján képzett pakolt-decimális értékét adja vissza. `Conversion_Error` hibajelzést kapunk, ha `Item` értéke negatív, `Format` pedig `Packed_Unsigned`.

```
function Valid (Item   : in Byte_Array;
                Format : in Binary_Format)
   return Boolean;
```

A függvény `True` értéket ad vissza, ha `Item` illeszkedik a `Format` paraméterhez. A függvényérték akkor lesz `False`, ha az `Item` által reprezentált érték kívül esik a `Num` típus értékkészletén.

```
function Length (Format : in Binary_Format)
   return Natural;
```

A függvény annak a legrövidebb `Byte_Array` tömbnek a hosszát adja vissza, amely elegendő a `Num` típus minden értékének `Format` szerinti reprezentálásához.

```
function To_Decimal (Item   : in Byte_Array;
                     Format : in Binary_Format)
   return Num;
```

Egy `Num` típusú értéket állít elő az `Item` paraméter `Format` szerinti értelmezése alapján. A tizedesjegyek számát `Num'Scale` határozza meg. A függvényhívás `Conversion_Error` kivételt vált ki, ha az `Item` által reprezentált szám kívül esik a `Num` típus értékkészletén.

```
    function To_Binary (Item   : in Num;
                        Format : in Binary_Format)
       return Byte_Array;
```

A függvény a kapott szám Format alapján képzett Byte_Array értékét adja vissza.

```
    function To_Decimal (Item : in Binary)      return Num;
    function To_Decimal (Item : in Long_Binary) return Num;
```

Ezek a függvények a COBOL nyelv belső bináris formátumából konvertálnak a Num típusba. Conversion_Error hibajelzést kapunk, ha az Item értéke kívül esik a Num típus értékkészletén.

```
    function To_Binary      (Item : in Num)
       return Binary;
    function To_Long_Binary (Item : in Num)
       return Long_Binary;

  end Decimal_Conversions;
```

Ezek a függvények az Ada egy numerikus értékét konvertálják át COBOL (belső) bináris formátumra. Ha Item értéke túl nagy a visszatérési típus számára, akkor Conversion_Error hibajelzést kapunk.

```
private
   ...
end Interfaces.COBOL;
```

A fenti interfész – a C nyelv támogatásához hasonlóan – a bevezetett típusokon keresztül lehetővé teszi COBOL eljárások Adából történő meghívását, meglévő COBOL programok Adából való paraméterezését.

Egy T rekord típus esetén a Convention(COBOL, T) pragma használatával jelezhetjük a fordítónak, hogy a típus értékeit COBOL-kompatíbilis formában kell reprezentálni.

Az egyes implementációk tartalmazhatnak további konstans értékeket a Display_Format, Binary_Format és Packed_Format típusokból, az adott környezetben támogatott COBOL implementáció alapján.

Tekintsünk egy példát az Interfaces.COBOL csomag használatára: Tegyük fel, hogy van egy PROG nevű COBOL programunk, melynek deklarációja a LINKAGE szekcióban az alábbi:

```
 01 Parameter-Area
    05 NAME   PIC X(20).
    05 SSN    PIC X(9).
    05 SALARY PIC 99999V99 USAGE COMP.
```

Tegyük fel továbbá, hogy a program valamilyen algoritmus alapján megváltoztatja a `SALARY` paraméter értékét. Ekkor a fenti COBOL programot Adából az alábbi módon hívhatjuk meg:

```
with Interfaces.COBOL;
procedure Test_Call is

   package COBOL renames Interfaces.COBOL;

   type Salary_Type is delta 0.01 digits 7;

   type COBOL_Record is
      record
         Name   : COBOL.Numeric(1..20);
         SSN    : COBOL.Numeric(1..9);
         Salary : COBOL.Binary;  -- Assume Binary = 32 bits
      end record;
   pragma Convention (COBOL, COBOL_Record);

   procedure Prog (Item : in out COBOL_Record);
   pragma Import (COBOL, Prog, "PROG");

   package Salary_Conversions is
      new COBOL.Decimal_Conversions(Salary_Type);

   Some_Salary : Salary_Type := 12_345.67;
   Some_Record : COBOL_Record :=
      (Name   => "Johnson, John       ",
       SSN    => "111223333",
       Salary => Salary_Conversions.To_Binary(Some_Salary));

begin
   Prog (Some_Record);
   ...
end Test_Call;
```

B.5. Kapcsolódás a Fortran nyelvhez

Az Ada Fortran nyelvvel való kapcsolata is két részből áll: egyrészt csomag szinten az `Interfaces.Fortran` csomag definiálja a Fortran típusainak megfelelő típusokat és a rájuk vonatkozó konverziókat, másrészt támogatja az `Import`, `Export` és `Convention` pragmákban a `Fortran` konvenció-azonosító használatát.

A csomag deklarációja a következő:

```
with Ada.Numerics.Generic_Complex_Types;
pragma Elaborate_All(Ada.Numerics.Generic_Complex_Types);
package Interfaces.Fortran is
   pragma Pure(Fortran);

   type Fortran_Integer is range implementation-defined;

   type Real             is digits implementation-defined;
   type Double_Precision is digits implementation-defined;

   type Logical is new Boolean;

   package Single_Precision_Complex_Types is
      new Ada.Numerics.Generic_Complex_Types (Real);

   type Complex is
      new Single_Precision_Complex_Types.Complex;

   subtype Imaginary is
      Single_Precision_Complex_Types.Imaginary;

   i : Imaginary renames Single_Precision_Complex_Types.i;
   j : Imaginary renames Single_Precision_Complex_Types.j;

   type Character_Set is
      implementation-defined character type;

   type Fortran_Character is array (Positive range <>)
      of Character_Set;
   pragma Pack (Fortran_Character);

   function To_Fortran (Item : in Character)
      return Character_Set;
   function To_Ada (Item : in Character_Set)
      return Character;

   function To_Fortran (Item : in String)
      return Fortran_Character;
   function To_Ada     (Item : in Fortran_Character)
      return String;
```

```
   procedure To_Fortran (Item   : in String;
                         Target : out Fortran_Character;
                         Last   : out Natural);

   procedure To_Ada (Item     : in Fortran_Character;
                     Target   : out String;
                     Last     : out Natural);

end Interfaces.Fortran;
```

A `Fortran_Integer`, `Real`, `Double_Precision`, `Logical`, `Complex`, és `Fortran_Character` típusok Fortran-kompatíbilisek.

A `To_Fortran` és `To_Ada` függvények az Ada `Character`, illetve `String` típusa és a Fortran `Character_Set`, illetve `Fortran_Character` típusa között konvertálnak. A `To_Fortran` és `To_Ada` eljárások hatása pedig analóg az `Interfaces.COBOL` csomagbeli eljárásokéval.

Példaként tekintsük egy Fortran-ban megírt mátrix invertálás egy Ada programban történő felhasználását.

```
with Interfaces.Fortran;
use Interfaces.Fortran;
procedure Ada_Application is

  type Fortran_Matrix is array (Integer range <>,
                                Integer range <>)
          of Double_Precision;
  pragma Convention (Fortran, Fortran_Matrix);
  procedure Invert (Rank : in Fortran_Integer;
                    X    : in out Fortran_Matrix);
  pragma Import (Fortran, Invert);

  Rank      : constant Fortran_Integer := 100;
  My_Matrix : Fortran_Matrix (1 .. Rank, 1 .. Rank);

begin

  ...
  My_Matrix := ...;
  ...
  Invert (Rank, My_Matrix);
  ...

end Ada_Application;
```

C. Függelék

Rendszerprogramozás

Ebben a fejezetben az Ada alacsony szintű programozást támogató lehetőségeit vizsgáljuk meg közelebbről. Ezekre általában valós idejű, beágyazott, osztott vagy információs rendszerek fejlesztése során lehet szükségünk.

C.1. A gépi utasítások elérése

A nyelv minden egyes megvalósításában találhatunk lehetőséget gépi kód beillesztésére vagy belső (intrinsic) alprogramok használatára (esetleg mindkettőre). Az Ada objektumok operandusként történő felhasználását implementációfüggő attribútumok támogatják.

A `System.Machine_Code` csomag típusainak és a fent említett attribútumoknak a leírását az implementációk dokumentációjában találhatjuk meg.

C.2. Megszakításkezelés

A megszakítás mindig a hardware vagy az operációs rendszer által észlelt eseményeket jelzi a program felé. Egy megszakítás előfordulása két részre bontható: generálás és átadás. A generálás az a hardware vagy operációs rendszer esemény, amely a megszakítást a program számára elérhetővé teszi. Az átadás az a tevékenység, amely során – válaszként a megszakítás előfordulására – meghívódik a program valamely része. A generálás és az átadás között a megszakítás függő állapotban van. Bizonyos megszakítások blokkolhatóak. Amikor egy megszakítás blokkolva van, akkor annak előfordulásai nem kerülnek átadásra. Egyes megszakítások foglaltak, azaz hozzájuk felhasználói kezelőprogram nem kapcsolható. A foglalt megszakítások halmaza implementációnként változó.

A nem foglalt megszakításokhoz programegységeket csatolhatunk. Ezek a megszakításkezelők a megszakítás-előfordulás átadásakor hajtódnak végre.

Amíg a kezelő a megszakításhoz van csatolva, addig annak a megszakításnak az összes átadott előfordulására meghívódik. A kezelő végrehajtása alatt a megszakítás blokkolt állapotban van.

Amíg a megszakítás blokkolva van, az előfordulásai nem kerülnek átadásra. Az ekkor generált megszakítás előfordulások az implementációtól függően vagy függő állapotba kerülnek, vagy elvesznek.

A megszakításkezelőből továbbadott kivételnek nincs hatása.

A nyelv egyes megvalósításainak dokumentációi tartalmazzák a megszakításokra vonatkozó további – implementációfüggő – információkat.

C.3. Védett kezelőeljárások

Adában az `Interrupt_Handler` és `Attach_Handler` pragmák használatával tehetjük az általunk írt megszakításkezelő eljárásokat védetté a megszakításokkal szemben. A pragmák szintaxisa:

```
pragma Interrupt_Handler(handler_name);

pragma Attach_Handler(handler_name, expression);
```

ahol `handler_name` egy paraméter nélküli eljárás azonosítója, `expression` pedig egy `Interrupts.Interrupt_ID` típusú kifejezés.

A fenti pragmákat csak azon a védett definíción belül használhatjuk, ahol annak az eljárásnak deklarációja található, amelyre hivatkoznak.

Ha egy eljárásra alkalmazzuk az `Interrupt_Handler` pragmát, akkor az az eljárás dinamikusan hozzácsatolható – akár egyidejűleg több – megszakításhoz is.

Az `Attach_Handler` pragmában megadott kifejezés a pragmát tartalmazó objektum inicializálásakor kiértékelődik, majd a megadott eljárás a kifejezés értékéül kapott megszakításhoz csatolódik.

Amikor a befoglaló objektum befejeződik, akkor a benne található összes olyan eljárás, amely megszakításhoz volt csatolva leválasztódik a megszakításról. Ha a kezelő az `Interrupts` csomag valamely eljárásán keresztül lett a megszakításhoz csatolva, akkor a megszakításhoz az alapértelmezett kezelés lesz hozzárendelve. Ha az eljárás az `Attach_Handler` pragmán keresztül lett a megszakításhoz csatolva, akkor az előző megszakításkezelő – vagy ennek hiányában az alapértelmezett kezelés – lesz visszaállítva.

Ha egy kezelő eljárás egy megszakításhoz van hozzárendelve, akkor minden az őt befoglaló védett objektumon végrehajtott védett művelet alatt a megszakítás blokkolódik.

C.4. Az Interrupts csomag

A nyelv minden implementációjában megtalálható az `Interrupts` és az `Interrupts.Names` csomag. Az `Ada.Interrupts` csomag deklarációja:

```
with System;
package Ada.Interrupts is
   type Interrupt_ID is implementation-defined;
   type Parameterless_Handler is
      access protected procedure;
```

Az `Interrupt_ID` egy implementációfüggő diszkrét típus a megszakítások azonosítására. A `Parameterless_Handler` a megszakításokhoz csatolható kezelőeljárások típusa.

```
   function Is_Reserved (Interrupt : Interrupt_ID)
      return Boolean;
```

Az `Is_Reserved` függvény pontosan akkor ad vissza `True` értéket, ha a paraméterként kapott megszakítás foglalt.

```
   function Is_Attached (Interrupt : Interrupt_ID)
      return Boolean;
```

Az `Is_Attached` függvény pontosan akkor ad vissza `True` értéket, ha a paraméterként kapott megszakításhoz van felhasználói kezelő-eljárás csatolva.

```
   function Current_Handler (Interrupt : Interrupt_ID)
      return Parameterless_Handler;
```

A `Current_Handler` függvény a megadott megszakításhoz hozzárendelt kezelőeljárást adja vissza. Ha a megszakításhoz nincs felhasználói kezelőeljárás hozzácsatolva, akkor a függvény az alapértelmezés szerinti kezelést jelölő értéket adja vissza. Ha ezzel az értékkel hívjuk meg az `Attach_Handler` vagy `Exchange_Handler` műveleteket (ld. a következő két eljárást), akkor visszaállítjuk az alapértelmezett megszakításkezelést.

```
   procedure Attach_Handler
      (New_Handler : in Parameterless_Handler;
       Interrupt   : in Interrupt_ID);
```

Az `Attach_Handler` eljárás a megadott megszakításkezelőt hozzácsatolja a megadott megszakításhoz. Ezzel felüldefiniálja az eddigi megszakításkezelést. Ha `New_Handler` értéke `null`, akkor az alapértelmezett megszakításkezelést állítja vissza. Ha `New_Handler` egy olyan védett eljárást jelöl, amelyre nem alkalmaztuk az `Interrupt_Handler` pragmát, akkor `Program_Error` hibajelzést kapunk, és a megszakítás kezelése nem változik.

```
procedure Exchange_Handler
   (Old_Handler : out Parameterless_Handler;
    New_Handler : in Parameterless_Handler;
    Interrupt   : in Interrupt_ID);
```

Az Exchange_Handler eljárás úgy működik mint az Attach_Handler, csak itt az Old_Handler paraméterben visszakapjuk az előző megszakításkezelőt.

```
procedure Detach_Handler
   (Interrupt : in Interrupt_ID);
```

A Detach_Handler eljárás visszaállítja a megadott megszakítás kezelését az alapértelmezés szerintire.

```
function Reference(Interrupt : Interrupt_ID)
   return System.Address;
```

A Reference függvény egy olyan címet ad vissza, amelynek segítségével egy taszk belépési pontot csatolhatunk a megadott megszakításhoz. Ha taszk belépési pontok megszakításokhoz (vagy a megadott megszakításhoz) rendelése nem megengedett, akkor Program_Error hibajelzést kapunk.

```
private
   ...
end Ada.Interrupts;
```

Az Is_Reserved és Reference függvények kivételével a csomag minden olyan művelete, amelynek paraméterlistájában van Interrupt_ID típusú érték Program_Error kivételt vált ki, ha a megadott megszakítás foglalt.

Ha az Attach_Handler, Detach_Handler, vagy Exchange_Handler műveletek valamelyikét olyan megszakításra alkalmazzuk, amelyhez statikusan – az Attach_Handler pragmán keresztül – csatoltunk kezelőeljárást, akkor Program_Error hibajelzést kapunk, és a megszakítás kezelése nem változik.

Az Ada.Interrupts.Names csomag deklarációja teljesen implementációfüggő, Interrupt_ID típusú konstansok deklarációit tartalmazza.

C.5. A Discard_Names pragma

A Discard_Names pragmával jelezhetjük a fordítónak, hogy bizonyos, a programban előforduló nyelvi egységek által elfoglalt memóriaterületet csökkentse. A pragma szintaxisa:

```
pragma Discard_Names[([On => ] local_name)];
```

A pragma csak konfigurációs pragmaként (paraméter nélkül, ekkor az összes lehetséges nyelvi egységre vonatkozik), vagy közvetlenül a deklarációs részben vagy a csomag specifikációban használható (ekkor a megadott nyelvi egységre, vagy ennek hiányában a pragmát követő összes nyelvi egységre vonatkozik). A `local_name` paraméter a nyelvi egység azonosítója.

A nyelvi egység lehet nem származtatott felsorolt altípus, nem származtatott jelölt (`tagged`) altípus, vagy kivétel. Ha a pragmát egy típusra alkalmazzuk, akkor annak minden leszármazott típusára is vonatkozik.

Ha a pragmát egy felsorolt típusra alkalmazzuk, akkor a `Wide_Image` és `Wide_Value` attribútumok szemantikája arra a típusra implementációfüggő (az `Image` és `Value` attribútumok szemantikája a `Wide_Image` és `Wide_Value` függvényében nem változik). Ugyanekkor a `Text_IO.Enumeration_IO` csomag szemantikája is implementációfüggő.

Ha a pragmát egy jelölt (`tagged`) típusra alkalmazzuk, akkor arra a típusra a `Tags.Expanded_Name` függvény szemantikája implementációfüggő.

Hasonlóan ha a pragmát egy kivételre alkalmazzuk, akkor arra a kivételre az `Exceptions.Exception_Name` függvény szemantikája a nyelv adott implementációjától függ.

C.6. Osztott elérésű változók

Osztott változók használatának vezérlésére az `Atomic`, `Atomic_Components`, `Volatile`, `Volatile_Components` reprezentációs pragmák használhatók.

Ezek segítségével az objektumokat atomivá, vagy azonnalivá tehetjük. Az atomi objektum minden lekérdező és megváltoztató művelete oszthatatlan egészként viselkedik. Az azonnali objektum minden lekérdező és megváltoztató művelete közvetlenül az objektum által elfoglalt memórián hajtódik végre. A pragmák szintaxisa:

```
pragma Atomic(local_name);

pragma Volatile(local_name);

pragma Atomic_Components(array_local_name);

pragma Volatile_Components(array_local_name);
```

Atominak deklarálhatunk egy típust, ha alkalmazzuk rá az `Atomic` pragmát. Egy objektum akkor lesz atomi, ha az alábbi három eset közül valamelyik fennáll:

- az objektumra alkalmaztuk az `Atomic` pragmát,
- az objektum egy olyan tömb eleme, amelyre az `Atomic_Components` pragmát alkalmaztuk,

- az objektum típusa atomi.

Azonnalinak deklarálhatunk egy típust, ha alkalmazzuk rá a `Volatile` pragmát. Egy objektum akkor lesz azonnali, ha az alábbi három eset közül valamelyik fennáll:

- az objektumra alkalmaztuk a `Volatile` pragmát,
- az objektum egy olyan tömb eleme, amelyre a `Volatile_Components` pragmát alkalmaztuk,
- az objektum típusa azonnali.

Ezen kívül minden atomi típus és objektum azonnali is, továbbá egy azonnali objektum minden komponense is azonnali.

C.7. Taszkok azonosítása

A nyelv az `Ada.Task_Identification` csomagot az alábbi interfésszel deklarálja:

```
package Ada.Task_Identification is
   type Task_ID is private;
   Null_Task_ID : constant Task_ID;
```

A `Task_ID` típus egy értéke egy létező taszkot azonosít. Ezalól egy kivétel van, a `Null_Task_ID` konstans érték, amely semmilyen taszkot nem azonosít. A `Task_ID` típus minden objektumának kezdeti értéke `Null_Task_ID`.

```
   function  "=" (Left, Right : Task_ID) return Boolean;
```

Az "=" operátor akkor és csak akkor ad vissza `True` értéket, ha `Left` és `Right` ugyanazt a taszkot azonosítja, vagy ha mindkettő `Null_Task_ID`.

```
   function  Image       (T : Task_ID) return String;
```

Az `Image` függvény egy a `T` taszkot azonosító, implementációfüggő sztringet ad vissza. Ha `T` `Null_Task_ID`, akkor a visszatérési érték üres sztring.

```
   function  Current_Task return Task_ID;
```

A `Current_Task` függvény a hívó taszkot azonosító értéket adja vissza. A függvény entry törzsből vagy megszakításkezelőből történő hívása korlátozott hibát (bounded error) okoz. Ekkor vagy `Program_Error` hibajelzést kapunk, vagy a függvényérték valamilyen implementációfüggő érték lesz.

```
    procedure Abort_Task   (T : in out Task_ID);
```

Az Abort_Task eljárás hatása a T Task_ID értékre megegyezik az abort utasítás hatásával a T által azonosított taszkra. T = Null_Task_ID esetén Program_Error hibajelzést kapunk.

```
    function  Is_Terminated(T : Task_ID) return Boolean;
    function  Is_Callable  (T : Task_ID) return Boolean;
```

Az Is_Terminated és Is_Callable függvények a T által azonosított taszk megfelelő attribútumának értékét adják vissza. T = Null_Task_ID esetén mindkét függvény Program_Error hibajelzést ad.

```
private
   ...
end Ada.Task_Identification;
```

Ha bármely ezen csomagbeli műveletnek olyan Task_ID értéket adunk át, amely által azonosított taszk már nem létezik, akkor a műveletek végrehajtása hibás.

Egy T taszk típusú objektumra alkalmazható a T'Identity attribútum, amely a T taszkot azonosító Task_ID értéket adja vissza.

Egy E belépési pont deklarációra alkalmazható az E'Caller attribútum, amely annak a taszknak az azonosítóját (Task_ID típusú) adja vissza, amelyet éppen kiszolgál. Ezt az attribútumot csak a deklarációhoz kapcsolódó törzsben vagy accept utasításban használhatjuk.

C.8. A Task_Attributes sablon csomag

A nyelv az Ada.Task_Attributes csomagot az alábbi interfésszel deklarálja:

```
with Ada.Task_Identification; use Ada.Task_Identification;
generic
   type Attribute is private;
   Initial_Value : in Attribute;
package Ada.Task_Attributes is

   type Attribute_Handle is access all Attribute;
```

Amikor a Task_Attributes sablon csomagot példányosítjuk, akkor minden még be nem fejeződött taszkhoz automatikusan létrejön egy az Attribute formális paraméternek megfelelő aktuális típusú objektum. Ez az objektum a taszk felhasználó által definiált attribútumaként viselkedik. A példányosítást követően indított taszkokhoz a létrehozáskor keletkezik egy ilyen objektum. Az objektum kezdeti értéke mindkét esetben Initial_Value lesz.

```
function Value(T : Task_ID := Current_Task)
  return Attribute;
```

A Value művelet visszaadja a T taszk ezen attribútumának értékét.

```
function Reference(T : Task_ID := Current_Task)
  return Attribute_Handle;
```

A Reference művelet a T taszk attribútum-objektumára mutató pointert ad vissza.

```
procedure Set_Value(Val : in Attribute;
                    T : in Task_ID
                        := Current_Task);
```

A Set_Value művelet végrehajtja a T taszk attribútumának megszüntetésével járó tevékenységeket, majd az attribútumnak értékül adja Val értékét.

```
procedure Reinitialize(T : in Task_ID := Current_Task);
```

A Reinitialize művelet ekvivalens a Set_Value művelet meghívásával úgy, hogy a Val paramétert az Initial_Value értékkel helyettesítjük.

```
end Ada.Task_Attributes;
```

Tasking_Error hibajelzést kapunk a csomag összes műveletétől, ha a T paraméter által azonosított taszk már befejeződött. T = Null_task_ID esetén Program_Error hibajelzést kapunk.

Hibás egy a Reference által visszaadott pointer értékére hivatkozni, ha a Reference művelet hívása óta végrehajtottunk egy Reinitialize hívást, vagy a taszk befejeződött.

A program végrehajtása akkor is hibás, ha bármely műveletnek olyan Task_ID típusú értéket adunk át, amely érték által hivatkozott taszk objektum már nem létezik.

Az egyes implementációk korlátozhatják az egy taszkhoz hozzárendelhető attribútumok maximális számát, egy attribútum méretét, vagy az attribútumok által elfoglalható memória maximális méretét.

D. Függelék

Valós idejű rendszerek

D.1. Taszk prioritások

A taszkprioritás egy egész szám, amely meghatározza, hogy mennyire sürgős a taszk futásához szükséges forrásokat a taszk rendelkezésére bocsátani. A több taszk között megosztott források mindig a legnagyobb prioritásúhoz vannak előbb hozzárendelve. Egy taszk alapprioritása a létrehozásakor kapott prioritás, amelyet a `Dynamic_Priorities.Set_Priority` (ld. később) művelettel megváltoztathatunk. Emellett egy taszknak mindig van egy aktív prioritása, amely az alapprioritást tükrözi, a más forrásokból származó öröklött prioritásokkal együtt. A prioritás öröklődés az a folyamat, amely során egy taszk vagy más egység (pl. védett objektum) prioritását használjuk fel egy másik taszk aktív prioritásának kiértékelésében (a továbbiakban bevezetésre kerülő pragmákban megadható kifejezés).

A párhuzamosan végrehajtandó programrészek közötti prioritások megadására a `Priority` és `Interrupt_Priority` pragmákat használhatjuk. A pragmák szintaxisa:

```
pragma Priority(expression);

pragma Interrupt_Priority[(expression)];
```

ahol `expression` egy egész típusú kifejezés.

A `Priority` pragma csak taszk definícióban, védett definícióban, vagy alprogram törzs deklarációs részében szerepelhet. Az `Interrupt_Priority` pragma csak taszk definícióban, vagy védett definícióban használható. További megszorítás, hogy legfeljebb egy ilyen pragma szerepelhet a fenti konstrukciókban.

Az alprogram törzs deklarációs részében előforduló `Priority` pragmának csak akkor van hatása, ha az alprogram a főprogramot definiáló eljárás. Ekkor a megadott kifejezésnek statikusnak kell lennie, és értékének bele kell esnie a `System.Priority` típus értékkészletébe. Ennek megfelelően a `System` csomag tartalmazza az alábbi deklarációkat:

```
subtype Any_Priority is Integer
   range implementation-defined;
subtype Priority is Any_Priority
   range Any_Priority'First .. implementation-defined;
subtype Interrupt_Priority is Any_Priority
   range Priority'Last+1 .. Any_Priority'Last;

Default_Priority : constant Priority :=
   (Priority'First + Priority'Last)/2;
```

A nyelv adott implementációja által támogatott prioritás-értékek teljes értékkészletét az `Any_Priority` altípus határozza meg. Ez a típus két részre oszlik: azok a prioritás értékek, amelyek elég nagyok ahhoz, hogy megszakításokat is blokkoljanak az `Interrupt_Priority` altípusba esnek. Az `Any_Priority` intervallum alsó részébe eső értékek alkotják a `Priority` altípus értékhalmazát.

A `Priority` és `Interrupt_Priority` pragmákban szereplő kifejezés értéke adja meg a prioritást. Ha az `Interrupt_Priority` pragmában nem adunk meg kifejezést, akkor a prioritás értéke `Interrupt_Priority'Last` lesz.

Egy taszk definíciójában található `Priority` vagy `Interrupt_Priority` pragmában levő kifejezés minden taszk objektumra kiértékelődik. A pragmákban szereplő kifejezés értéke `Priority` pragma esetén `Priority` altípusra míg az `Interrupt_Priority` pragma esetén `Any_Priority` altípusra konvertálódik. Ezáltal a prioritás érték hozzárendelődik ahhoz a taszk objektumhoz, amelynek definíciója a pragmát tartalmazza.

Hasonlóan, ha a pragma a főprogram deklarációs részében szerepel, akkor a prioritás érték a környezeti taszkhoz lesz hozzárendelve.

Egy taszk kezdeti alapprioritását a `Priority` és `Interrupt_Priority` pragmák határozzák meg. Ezek hiányában egy taszk alapprioritása az őt létrehozó taszk – a létrehozás időpontjában meglevő – alapprioritása lesz. Ha a főprogramra nem vonatkozik Priority pragma, akkor a környezeti taszk prioritása `System.Default_Priority` lesz.

Egy tetszőleges időpillanatban egy taszk aktív prioritása a taszk által öröklött prioritások maximuma. Egy nem érvényes taszk esetén a prioritás öröklődés forrása a taszk alapprioritása. A prioritás öröklődésnek lehetnek további forrásai az alábbi esetekben:

- Aktiválás során az aktivált taszk örökli az aktiváló taszk aktív prioritását.
- Randevúk esetén a belépési pontot elfogadó taszk a hívó taszk aktív prioritását örökli.
- Egy védett objektum védett végrehajtása során a taszk a védett objektum legfelső prioritását örökli.

Mindhárom fenti esetben a prioritás öröklés megszűnik, amint az öröklődés feltétele már nem teljesül.

D.2. Prioritás ütemezés

A továbbiakban az egyszerre készen álló taszkok közüli választásra vonatkozó szabályokat tekintjük át.

D.2.1. A taszk ütemezési modell

Egy taszk akkor indul el, ha készen áll, és a végrehajtásához szükséges források a rendelkezésére állnak. A processzorok megosztása a taszkok között az aktív prioritás alapján történik.

A taszk indítás az a választási folyamat, amely meghatározza, hogy mely készen álló taszk kerüljön végrehajtásra egy processzoron. Erre a választásra egy taszk futása alatt az ún. taszk ütemezési pontokban kerül sor. Egy taszk akkor ér el egy ilyen pontot, ha blokkolódik, vagy késszé válik. Hasonlóan egy `accept` utasítás végrahajtása, vagy a taszk terminálása is ilyen taszk ütemezési pont.

A taszk indítási technikákat a *kész-sorok*, a taszk-állapotok és a taszk előzés fogalmán keresztül fogjuk megfogalmazni. Kész-sornak nevezzük a készen álló taszkok rendezett listáját. A listában az első pozíciót a sor fejének, az utolsót a sor végének nevezzük. Egy taszk akkor áll készen, ha egy kész sorban áll, vagy éppen fut. Az ütemezés modellje a következő:

Minden processzornak minden prioritás értékhez van egy kész-sora, és minden időpillanatban minden kész-sor pontosan azokat a készen álló – és az adott prioritással rendelkező – taszkokat tartalmazza, amelyek nem futnak, és az adott processzor és más elérhető források felhasználásával végrehajthatók. Egy taszk egyszerre több processzor kész-során is szerepelhet.

Minden processzorhoz ugyanakkor hozzátartozik egy futó taszk is. Amikor ez a taszk elér egy ütemezési pontot, ki kell választani, hogy a továbbiakban mely taszk fusson az adott processzoron. Ez a taszk a nem üres kész-sorok közül annak a feje, amely a legnagyobb prioritáshoz tartozik. Ez a taszk kikerül az összes olyan kész-sorból, amelyhez eddig hozzátartozott.

Ha egy taszkhoz kötött forrás ideigelenesen az eredeti taszk helyett más taszkhoz is hozzáköthető, akkor a forrást előzés-támogató forrásnak nevezzük. A processzorok például ilyen előzés-támogató források, míg egy védett objektum elérése nem előzés-támogató.

Amikor egy nagyobb prioritású taszk kerül végrehajtásra egy processzoron, akkor az eredeti taszk bekerül a megfelelő kész-sor(ok)ba, és azt mondjuk hogy az új taszk megelőzte az eredetileg futót.

Taszk ütemezési pontok lehetnek továbbá még azok az esetek is, amikor az aktuálisan futó taszknál nagyobb prioritású kész-sorok közül valamelyik nem üres, vagy a használt ütemezési technika a futó taszkot visszakényszeríti egy kész-sorba.

D.2.2. A szabványos taszk ütemező technika

Taszk ütemező technikának nevezzük azt a szabályrendszert, amely a fentiekben leírt ütemezési modell részleteit határozza meg. Ezek a szabályok definiálják, hogy egy taszk mikor kerül be egy kész sorba, mikor törlődik egy kész-sorból, vagy akár azt, hogy a taszk az aktív prioritásának megfelelő kész-sor elejére, vagy végére kerül-e be. A taszk ütemezési technikát a `Task_Dispatching_Policy` konfigurációs pragmán keresztül választhatjuk ki. A pragma szintaxisa:

```
pragma Task_Dispatching_Policy(policy_identifier);
```

ahol `policy_identifier` egy ütemező technika azonosító. A nyelv által definiált azonosító a `FIFO_Within_Priorities`, de az egyes implementációk más technikákat is megvalósíthatnak más azonosítókkal. Ezt az egyes implementációk dokumentálják.

A `FIFO_Within_Priorities` technika az alábbi szabályrendszerből áll:

- ha egy blokkolt taszk kész állapotúvá válik, bekerül az aktív prioritásának megfelelő kész-sor végére,
- ha egy nem futó készen álló taszk aktív prioritása megnő, akkor kikerül a régi aktív prioritásának megfelelő kész-sorból, és bekerül az új aktív prioritásának megfelelő kész-sor végére,
- ha egy nem futó készen álló taszk aktív prioritása lecsökken, akkor kikerül a régi aktív prioritásának megfelelő kész-sorból, és bekerül az új aktív prioritásának megfelelő kész-sor elejére,
- amikor egy futó taszk alapprioritásának beállítása érvényessé válik, a taszk bekerül az aktív prioritásának megfelelő kész-sor végére,
- amikor egy taszk olyan delay-utasítást hajt végre, amely nem eredményez blokkolt állapotot, a taszk bekerül az aktív prioritásának megfelelő kész-sor végére,
- ha egy taszk megelőz egy másik taszkot, akkor a megelőzött taszk bekerül az aktív prioritásának megfelelő kész-sor elejére.

A fenti esetek mind taszk ütemezési pontnak számítanak.

D.3. Maximált prioritás kizárási technika

A védett objektumok zárolásának szabályait a kizárási technika határozza meg. Ezek a szabályok határozzák meg, hogy a védett objektumokhoz rendelünk-e prioritásokat, és ha igen, akkor mi a kapcsolat ezen prioritások és a taszk prioritások között. Azt is a kizárási technika határozza meg, hogy milyen hatással van a taszkra egy védett művelet végrehajtása. A kizárási technikát a `Locking_Policy` pragmán keresztül választhatjuk ki, melynek szintaxisa:

```
pragma Locking_Policy(policy_identifier);
```

ahol `policy_identifier` egy kizárási technika azonosító. A nyelv egyetlen ilyen azonosítót definiál, de az egyes implementációk bevezethetnek további azonosítókat is (ezeket az implementációk dokumentálják).

A nyelv által definiált – `Ceiling_Locking` azonosítójú – kizárási technika az alábbi szabályrendszerből áll:

- Minden védett objektumnak van egy prioritás maximum értéke, amelyet a `Priority` vagy az `Interrupt_Priority` pragmán keresztül adhatunk meg. Ez az érték felső korlát azon taszkok aktív prioritására, amelyek meghívják ennek a védett objektumnak a védett műveleteit.

- A `Priority` és `Interrupt_Priority` pragmákban szereplő kifejezések az objektum létrehozásakor értékelődnek ki, majd a megfelelő típusúra konvertálódnak. Ez az érték lesz az objektum maximált prioritása.

- Ha egy védett objektum definíciójában szerepel `Interrupt_Handler` vagy `Attach_Handler` pragma, de az `Interrupt_Priority` pragmán keresztül nincs megadva prioritás, akkor az objektum maximált prioritása implementációfüggő, a `System.Interrupt_Priority` altípusba eső érték lesz.

- Ha egy védett objektum definíciójában nem adunk meg prioritást a `Priority` vagy az `Interrupt_Priority` pragmán keresztül, és a definícióban nem szerepel `Interrupt_Handler` ill. `Attach_Handler` pragma, akkor `System.Priority'Last` lesz az objektum maximált prioritása.

- Amíg egy taszk egy védett műveletet hajt végre, megörökli a megfelelő objektum maximált prioritását.

Ha egy taszk meghív egy védett műveletet, és a taszk prioritása nagyobb mint a védett objektum maximált prioritása, akkor `Program_Error` hibajelzést kapunk.

D.4. Belépési pont besorolási technikák

Azt a szabályrendszert, amely egy belépési pontra várakozó taszkok sorrendjét meghatározza, besorolási technikának nevezzük. A besorolási technika a `Queuing_Policy` konfigurációs pragmán keresztül adható meg, melynek szintaxisa:

```
pragma Queuing_Policy(policy_identifier);
```

ahol `policy_identifier` egy besorolási technika azonosító. A nyelv két azonosítót (`FIFO_Queuing` és `Priority_Queuing`) definiál, de az egyes implementációk más azonosítókat – és persze technikákat – is bevezethetnek. Ha a programban sehol sem szerepel `Queuing_Policy` pragma, akkor a besorolási technika `FIFO_Queuing` lesz.

A `Priority_Queuing` besorolási technika szabályrendszere:

- A belépési pont hívások a hívások prioritása alapján vannak sorbaállítva. A hívás prioritásának kezdeti értéke a hívó taszk aktív prioritása a hívás időpontjában. Az azonos prioritású hívások közötti sorrendet a hívások időpontja határozza meg (FIFO sorrend).

- Miután egy hívás bekerül egy sorba, a hívó taszk aktív prioritásának megváltozása nincs hatással a hívás prioritására, kivéve azt az esetet, amikor a taszk alapprioritása is megváltozik.

- Ha a hívó taszk alapprioritása megváltozik, akkor a taszk az új aktív prioritása alapján újra besorolódik és az eredeti bejegyzés törlődik a várakozási sorból

- Ha egy védett objektum belépési korlátjának (entry barrier) több feltétele is teljesül, és a hozzájuk tartozó sorok közül legalább kettő nem üres, akkor a magasabb prioritású hívás lesz kiválasztva. Ha több azonos prioritású ilyen hívás van, akkor az kerül kiválasztásra, amely a védett definícióban korábban szerepel. Az azonos belépési pont család tagjai közül a kisebb indexűre esik a választás.

- Ha kettő vagy több nyitott késleltetési alternatíva lejárati ideje megegyezik és nincs más nyitott alternatíva-elfogadás, akkor a programszövegben korábban szereplő kerül végrehajtásra.

- Ha egy szelektív accept több alternatívájához is tartozik nem üres várakozási sor, akkor arra az alternatívára kerül a vezérlés, amelyhez tartozó sor fejének nagyobb a prioritása. Ha ebben az esetben több sor is eleget tesz a fenti feltételnek, akkor a programszövegben először előforduló lesz kiválasztva.

D.5. Dinamikus prioritáskezelés

Egy taszk alapprioritását futási időben le lehet kérdezni, és meg is lehet változtatni. Erre a célra szolgál a nyelv által definiált Dynamic_Priorities csomag:

```
with System;
with Ada.Task_Identification;
package Ada.Dynamic_Priorities is

   procedure Set_Priority(
      Priority : in System.Any_Priority;
      T : in Ada.Task_Identification.Task_ID :=
             Ada.Task_Identification.Current_Task);
```

A Set_Priority eljárás a megadott taszk alapprioritását a Priority paraméter értékére állítja. A beállítás addig nem történik meg, amíg a taszk védett műveletet hajt végre. Ha a taszk már terminált, akkor az eljárásnak nincs hatása.

```
   function Get_Priority(
      T : Ada.Task_Identification.Task_ID :=
          Ada.Task_Identification.Current_Task)
      return System.Any_Priority;

end Ada.Dynamic_Priorities;
```

A Get_Priority függvény visszaadja a T taszk aktuális alapprioritását. Tasking_Error hibát jelez, ha a taszk már terminált.

Mindkét fenti művelet Program_Error kivételt vált ki, ha T megegyezik a Null_Task_ID konstanssal.

D.6. Monoton idő

A nyelv a Real_Time csomagot definiálja az időpontok és időtartamok kezelésére.

```
package Ada.Real_Time is

  type Time is private;
  Time_First : constant Time;
  Time_Last : constant Time;
  Time_Unit : constant :=
     implementation-defined-real-number;
```

A Time típus értékei egy ideális idővonal szegmenseit reprezentálják. A típusértékek halmaza és a matematikai egészek egy intervalluma kölcsönösen egyértelműen leképezhető egymásra.

A I egész számhoz kapcsolódó Time érték az $[E + I * \texttt{Time_Unit}, E + (I + 1) * \texttt{Time_Unit})$ balról zárt jobbról nyílt időintervallumot reprezentálja, ahol Time_Unit egy implementációfüggő valós szám, E pedig egy nem specifikált kiindulási időpont, amely a Time típus összes értékére ugyanaz (epoch). Time_Unit a Time típus által reprezentálható legkisebb időegység másodpercekben kifejezve.

```
type Time_Span is private;
Time_Span_First : constant Time_Span;
Time_Span_Last : constant Time_Span;
Time_Span_Zero : constant Time_Span;
Time_Span_Unit : constant Time_Span;
```

A Time_Span típus értékei egy valós időtartamot reprezentálnak. A típusértékek és a matematikai egészek egy intervalluma között kölcsönősen egyértelmű leképezés adható. Az I egész számhoz kapcsolódó Time_Span érték az $I * \texttt{Time_Unit}$ időtartamot reprezentálja.

A Time_Span_Unit konstans értéke két egymást követő Time érték közötti különbség. Ez ugyanakkor megegyezik a legkisebb pozitív Time_Span értékkel is. A Time_Unit és Time_Span_Unit konstansok ugyanazt a valós időtartamot reprezentálják.

```
Tick : constant Time_Span;
function Clock return Time;
```

Óraütésnek nevezzük azt a valós időintervallumot, amely alatt az óraidő (a Clock függvény visszatérési értéke) konstans marad. A Tick konstans ezeknek az időintervallumoknak az átlagát tartalmazza. A Clock függvény az E (epoch) kezdő időpont óta eltelt időt adja vissza.

```
function "+" (Left : Time; Right : Time_Span) return Time;
function "+" (Left : Time_Span; Right : Time) return Time;
function "-" (Left : Time; Right : Time_Span) return Time;
function "-" (Left : Time; Right : Time) return Time_Span;

function "<" (Left, Right : Time) return Boolean;
function "<="(Left, Right : Time) return Boolean;
function ">" (Left, Right : Time) return Boolean;
function ">="(Left, Right : Time) return Boolean;

function "+" (Left, Right : Time_Span) return Time_Span;
```

```
function "-" (Left, Right : Time_Span) return Time_Span;
function "-" (Right : Time_Span) return Time_Span;
function "*" (Left : Time_Span; Right : Integer)
   return Time_Span;
function "*" (Left : Integer; Right : Time_Span)
   return Time_Span;
function "/" (Left, Right : Time_Span)
   return Integer;
function "/" (Left : Time_Span; Right : Integer)
   return Time_Span;

function "abs"(Right : Time_Span) return Time_Span;

function "<" (Left, Right : Time_Span) return Boolean;
function "<="(Left, Right : Time_Span) return Boolean;
function ">" (Left, Right : Time_Span) return Boolean;
function ">="(Left, Right : Time_Span) return Boolean;
```

A fenti függvények az imént bevezetett típusok matematikai és összehasonlító műveleteit definiálják.

```
function To_Duration (TS : Time_Span) return Duration;
function To_Time_Span (D : Duration) return Time_Span;
```

A `To_Duration` függvény a `TS` paraméter értékét `Duration` típusúvá, míg a `To_Time_Span` függvény a `D` paraméter értékét `Time_Span` típusúvá konvertálja. Mindkét műveletben az eredmény a legközelebbi ábrázolható értékre lesz kerekítve. `To_Duration(Time_Span_Zero)` értéke `0.0`, míg fordított irányban `To_Time_Span(0.0)` értéke `Time_Span_Zero`.

```
function Nanoseconds  (NS : Integer) return Time_Span;
function Microseconds (US : Integer) return Time_Span;
function Milliseconds (MS : Integer) return Time_Span;
```

A `Nanoseconds`, `Microseconds`, és `Milliseconds` függvények a megadott paramétert `Time_Span` értékké konvertálják az `NS`, `US`, és `MS` értékeket nanoszekundumban, mikroszekundumban ill. milliszekundumban megadott időtartamoknak tekintve. Mindhárom műveletben az eredmény a legközelebbi ábrázolható értékre van kerekítve.

```
type Seconds_Count is range implementation-defined;
```

A `Seconds_Count` típus értékei az epoch kiindulási időpont óta eltelt idő másodpercben mért értékét reprezentálja.

```
  procedure Split(T  : in Time;
                  SC : out Seconds_Count;
                  TS : out Time_Span);
  function Time_Of(SC : Seconds_Count;
                   TS : Time_Span) return Time;

private
   ...
end Ada.Real_Time;
```

A `Split` és `Time_Of` műveleteket úgy definiáljuk, hogy a `Time`, `Time_Span` és `Seconds_Count` típus értékeit egész számként fogjuk fel. Ekkor a `Split` művelet `SC` és `TS` értékét úgy állítja be, hogy az alábbi két feltétel teljesül:

```
      T*Time_Unit = SC*1.0+TS*Time_Unit
             0.0 <= TS*Time_Unit < 1.0.
```

A `Time_Of(SC, TS)` hívás által visszaadott `T Time` értékre teljesül az alábbi egyenlőség:

```
     T*Time_Unit = SC*1.0+TS*Time_Unit.
```

D.7. Szinkron taszk vezérlés

A nyelv által definiált `Synchronous_Task_Control` csomag egy átlátszatlan szemafor típust valósít meg. A csomag definíciója:

```
package Ada.Synchronous_Task_Control is

  type Suspension_Object is limited private;
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object)
     return Boolean;
  procedure Suspend_Until_True(S : in out Suspension_Object);
private
     ...
end Ada.Synchronous_Task_Control;
```

A `Suspension_Object` típus egy objektumának két látható állapota van: `True` és `False`. Az állapot kezdeti értéke `False`. A `Set_True` és `Set_False` műveletek – amelyek az állapotot állítják be `True` ill. `False` értékre – atomiak egymáshoz és a `Suspend_Until_True` művelethez viszonyítva.

A `Current_State` függvény az objektum aktuális állapotát adja vissza.

A Suspend_Until_True eljárás egészen addig blokkolja a hívó taszkot, amíg az S objektum állapota True lesz. Ekkor a taszk kész állapotba kerül és az objektum False állapotú lesz. Az eljárás hívása Program_Error kivételt vált ki, ha a paraméter objektum már várakoztat egy másik taszkot.

D.8. Aszinkron taszk vezérlés

Az Asynchronous_Task_Control csomag szolgálatatásain keresztül aszinkron taszk vezérlést valósíthatunk meg. Ez a csomag egy elméleti felfüggesztési prioritás értéket használ a taszk felfüggesztett állapotának jelzésére.

```
with Ada.Task_Identification;
package Ada.Asynchronous_Task_Control is
  procedure Hold(T : in Ada.Task_Identification.Task_ID);
  procedure Continue(T : in Ada.Task_Identification.Task_ID);
  function Is_Held(T : Ada.Task_Identification.Task_ID)
   return Boolean;
end Ada.Asynchronous_Task_Control;
```

Minden processzorhoz van egy elméleti üres taszk, ami mindig kész állapotban van. Ennek alapprioritása a System.Any_Priority'First értéknél kisebb. A felfüggesztési prioritás egy az üres taszk prioritásánál is kisebb konstans egész.

A Hold művelet a T taszk állapotát felfüggesztetté teszi: ekkor a taszk alapprioritása helyett a felfüggesztési prioritás szolgál prioritás-öröklődési forrásként.

A Continue művelet hatására a T taszk újra felfüggesztetlenné válik, és aktív prioritása újraszámolódik az alapprioritást alapul véve.

Az Is_Held művelet akkor és csak akkor ad vissza True értéket, ha a paraméter T taszk felfüggesztett állapotban van.

A fenti három művelet mindegyike Tasking_Error kivételt vált ki, ha a T taszk már terminált. Ha T értéke Null_Task_ID, akkor Program_Error hibajelzést kapunk.

E. Függelék

Osztott rendszerek

Ebben a fejezetben az Ada azon eszközeivel ismerkedünk meg, amelyek segítségével több, egymással kooperatívan működő partícióból álló, úgynevezett elosztott programokat készíthetünk. Egy partíció könyvtári egységek halmaza. A partíciók mindegyike külön címtérben fut, akár külön számítógépen is. Végrehajtásuk független, kivéve amikor kommunikálnak egymással. Ebben a részben azokkal a direktívákkal ismerkedünk meg, amelyekkel a partíciók közötti kommunikáció megteremthető és szabályozható.

A továbbiakban osztott rendszernek az elosztott programot futtató hardvert, illetve az azt működtető rendszerprogramokat nevezzük. Egy osztott rendszerben az erőforrásokat két csoportba sorolhatjuk. Az egyik csoport a *feldolgozó egység*ek, olyan erőforrások, amelyek mind számítási, mind tárolási képességekkel rendelkeznek. A másik csoportba a *tároló egység*ek tartoznak, amelyek csak adattárolási képességgel rendelkeznek és amelynek tárólóhelyeit egy vagy több feldolgozó egység is elérheti. Az egységek lehetnek egymástól akár nagy földrajzi távolságra ám az is elképzelhető, hogy valójában egyetlen egyprocesszoros rendszer autonóm processzei. Az osztott rendszernek mindezeken túl részét képezik még azok a kapcsolatok, kommunikációs csatornák is, amelyek ezeket az egységeket összekapcsolják.

E.1. Partíciók

Egy párhuzamos Ada nyelvű program autonóm részei a partíciók. A partíciók egy elosztott programban lehetnek *aktív*ak vagy *passzív*ak. Az aktív partíciók önálló végrehajtási szállal, esetleg szálakkal rendelkeznek. Minden aktív partíció rendelkezik egy névtelen környezeti taszkkal, amely a legkülső végrehajtási szintet jelenti. A taszk végrehajtása során történik a partíciót alkotó könyvtári egységek kiértékelése, majd a partíció főprogramjának elindítása, ha ilyen létezik. A partíció végrehajtása így a környezeti taszkjának végrehajtását jelenti. Egy aktív partíció akkor terminál, amikor a partíció környezeti taszkja terminál. A partíció nem-hozzáférhetővé válik, ha ter-

minál vagy abortál. Ez utóbbi a környezeti taszk abortálásakor következik be.

A passzív partícióknak nincs saját vezérlő száluk, könyvtári egységeik előre kiértékeltek, alprogramjaik és adataik egy vagy több aktív partíció számára hozzáférhetőek. Azt az eljárást, melynek során egy elosztott program partíciói leképeződnek egy adott osztott rendszer egységeire, a program partíciói *konfigurálásá*nak nevezzük. A partíciók konfigurálásakor az aktív partíciókat feldolgozó egységekre kell leképezni, míg a passzív partíciók tároló egységekre is képezhetőek. Egy partíció pontosan egy egységre képeződik ám az adott implementáció megengedheti, hogy több partíció képződjön egyazon egységre. Ilyenkor a használható ütemezési stratégiát, prioritáskezelést és a megosztott erőforrások menedzselésének módját az adott implementáció szabadon választhatja meg.

Egy könyvtári egység kiértékelésre kerül minden őt tartalmazó partíció kiértékelésekor. Ha a könyvtári egységnek van valamilyen állapota (például lokális változói), úgy az állapotnak egymástól független másolatai jönnek létre minden partícióban. Legyen `D` egy könyvtári szintű deklaráció, kivéve a `pure` könyvtári egységek deklarációit. Ekkor

`D'Partition_ID` jelöli az a `universal_integer` típusú értéket, amely azt a partíciót azonosítja, melyben az adott könyvtári egység kiértékelése megtörtént. Ha `D` egy távolról hívható könyvtári egység interfészének deklarációja, akkor az attribútum azon partíció azonosítóját adja, amelyben `D` törzse kiértékelésre került. Az attribútum értéke olyan, az adott implementációtól függő, alprogramok paramétereként használható, amelyek segítségével információkat kérhetünk az egyes partíciókról.

E.2. A könyvtári egységek osztályozása

A könyvtári egységeket osztályozhatjuk aszerint, hogy milyen szerepet játszanak egy elosztott programban. Az egyes kategóriákhoz különböző megszorítások társulnak, amelyek biztosítják, hogy az elosztott program szemantikája nagyjából megfeleljen egy nem elosztotténak.

A kategorizációs pragmák olyan, könyvtári egységekre vonatkozó direktívák, amelyek megszorítják a deklarációkat, alegységeket vagy szemantikus függőségeket a könyvtári egységben amelyre vonatkoznak. Az ilyen könyvtári egységeket *kategorizáltnak* nevezzük. Kategorizációs pragmák a `Shared_Passive`, `Remote_Types` és `Remote_Call_Interface`, és ilyennek tekinthető a `Pure` direktíva is. Ezen direktívák hatáskörébe eső könyvtári egységeket rendre osztott passzív-, távoli típus-, távoli hívás interfész-, illetve pure könyvtári egységnek nevezzük.

A fenti kategóriák egy hierarchiába szerveződnek oly módon, hogy egy adott kategóriába eső könyvtári egység csak a hierarchiában felette álló vagy vele megegyező kategóriájú egységektől függhet szemantikusan. Ez alól kivételt jelent a távoli típus- és távoli hívás interfész egységek törzse, amely nincs korlátozva. A hierarchia ennek megfelelően a következőképpen alakul:

Pure Csak más pure könyvtári egységektől függhet.

Osztott passzív Osztott passzív- vagy pure egységektől függhet.

Távoli típus A könyvtári egység deklarációja csak távoli típus vagy felette levő kategóriákba eső egységektől függhet. A könyvtári egység törzsére nincs megszorítás.

Távoli hívás interfész A deklaráció csak távoli hívás interfészektől és a hierarchiában felettük levő egységektől függhet. A törzsre nincs megszorítás.

Normál Nincs megszorítás.

E.2.1. Osztott passzív könyvtári egységek

Az osztott passzív könyvtári egységek aktív partíciók között megosztott globális adatok menedzselésére használatosak. Egy osztott passzív egységre vonatkozó megszorítások meggátolják, hogy az egyik aktív partíció adatait vagy taszkjait egy másik aktív partíció elérhesse az osztott passzív partícióban deklarált objektumok implicit referenciáin keresztül. A direktíva használata:

`pragma Shared_Passive`[(< *könyvtári egység neve* >)];

A könyvtári egységre az alábbi megszorítások kell, hogy teljesüljenek:

- Előre kiértékelhető kell, hogy legyen.
- Szemantikusan csak más osztott passzív-, illetve pure könyvtári egységektől függhet.
- Nem tartalmazhatja olyan könyvtári szintű mutató típus deklarációját amely osztályszintű típusra, taszk típusra, vagy olyan védett típusra hivatkozik, amelyhez tartozik entry-deklaráció.

Az osztott passzív könyvtári egységet hozzá kell rendelni az elosztott program valamely partíciójához, ám nem szükséges egy partíció fordítási egységeitől szemantikusan függő osztott passzív könyvtári egységeket az adott partícióba belefoglalni. Általában ezek a könyvtári egységek egy önálló passzív partícióba kerülnek.

E.2.2. Távoli típus könyvtári egységek

A távoli típus könyvtári egységek az aktív partíciók közötti kommunikáció céljait szolgáló adattípusok definiálására használatosak. Az egyik legfontosabb alkalmazásuk, hogy a távoli hívás interfész egységekben definiált alprogramok paramétereinek típusait ilyen egységekben kell definiálni. A megfelelő direktíva használata:

`pragma Remote_Types`[(< *könyvtári egység neve* >)];

A könyvtári egységre az alábbi megszorításoknak kell teljesülniük:

- Előre kiértékelhető kell, hogy legyen.
- Szemantikusan csak más távoli típus-, osztott passzív-, illetve pure könyvtári egységektől függhet.
- A könyvtári egység definíciójának látható részében nem szerepelhet változódeklaráció.
- A könyvtári egység definíciójának látható részében deklarált típusoknak ha van nem távoli mutató típusú része, akkor az adott mutató típushoz a felhasználónak kell definiálnia a `Read` és `Write` attribútumokat.

Azokat a mutató típusokat, amelyeket valamely távoli típus-, illetve távoli hívás interfész egységekben deklarálunk, távoli (remote) mutató típusnak nevezzük. Ezek a típusok csak alprogramokra vagy osztályszintű korlátozott átlátszatlan típusokra vonatkozhatnak.

Az alprogramra vonatkozó távoli mutató típusoknak a következő kritériumoknak kell megfelelniük:

- Alprogramra vonatkozó távoli mutató típusú értéket csak más alprogramra vonatkozó távoli mutató típusra lehet konvertálni (természetesen az egyéb, típuskonverzióra vonatkozó szabályok betartásával).
- Az `Access` attribútummal csak távoli alprogram esetén képezhető alprogramra vonatkozó távoli mutató típusú referencia.

Osztályszintű korlátozott átlátszatlan típusokra vonatkozó távoli mutató típusokra az alábbi megkötések érvényesek:

- A hivatkozott korlátozott átlátszatlan típus primitív műveleteinek a dinamikus összekapcsolást vezérlő paramétere csak mutató paraméter lehet, a nem vezérlő formális paraméterekhez pedig szükségesek a megfelelő `Read` és `Write` attribútumok.

- Osztályszintű típusra vonatkozó távoli mutató típusú értéket csak más osztályszintű típusra vonatkozó távoli mutató típusra lehet konvertálni (természetesen az egyéb, típuskonverzióra vonatkozó szabályok betartásával).
- Osztályszintű típusra vonatkozó távoli mutató típusú értékre dereferencia csak alprogramhíváskor alkalmazható, amelyben a hivatkozott objektum a hívandó alprogramot dinamikusan meghatározó paraméter.
- Távoli mutató típusú érték csak `Access` vagy `Unchecked_Access` attribútumok használatával keletkezhetnek, allokátor nem hívható távoli mutató típusra, így a `Storage_Pool` és `Storage_Size` attribútumok sem használhatóak rájuk.
- Távoli mutató típus nem használható aktuális paraméterként sablon példányosításakor, ha a formális paraméter mutató típusú.

Megjegyzés

Távoli típus könyvtári egységeknek nem kell pure-nak lenniük. A bennük definiált típusokban használhatunk több szintű indirekciót mutató típusok segítségével. A programozó által definiált `Read` és `Write` attribútumok segítségével történik aztán az adatok továbbítása az aktív partíciók között.

E.2.3. Távoli hívás interfész könyvtári egységek

A távoli hívás interfész segítségével olyan alprogramokat definiálhatunk, amelyek egyetlen aktív partícióban helyezkednek el ám más aktív partíciók is hívhatják őket úgynevezett távoli alprogramhívás formájában. Távoli hívás interfészt az alábbi direktíva használatával hozhatunk létre:

`pragma Remote_Call_Interface`[(< *könyvtári egység neve* >)];

Egy távoli hívás interfész könyvtári egységnek a következő feltételeknek kell megfelelnie:

- Előre kiértékelhetőnek kell lennie.
- Szemantikusan csak pure-, osztott passzív-, távoli típus- és más távoli hívás interfész könyvtári egységektől függhet.
- Nem tartalmazhat változódeklarációt.
- Nem szerepelhet benne korlátozott típus deklarációja.
- Nem tartalmazhat beágyazott sablon deklarációt.

- Nem tartalmazhat `inline` alprogram-deklarációt.
- Nem szerepelhet benne olyan alprogram- vagy alprogramra vonatkozó mutató típus deklarációja, amelynek szignatúrájában mutató paraméter szerepel[1].
- Csak olyan korlátozott típus használható formális paraméter típusaként, amelynek van a programozó által definiált `Read` és `Write` attribútuma.

Egy távoli hívás interfész csomagnak minden nyilvános (public) gyermeke is távoli hívás interfész csomag. Egy távoli hívás interfész könyvtári egység egy elosztott programban csak egyetlen partícióhoz rendelhető és ha vannak leszármazottai akkor azokat ugyanazon partícióhoz kell rendelni. Mivel azoknak a fordítási egységeknek, amelyek szemantikusan függenek egy távoli hívás interfésztől, annak csupán a specifikációjára van szükségük, így elegendő csak azt szerepeltetni az adott fordítási egységhez rendelt partícióban, a távoli hívás interfész törzsét pedig elegendő az őt tartalmazó partícióban elhelyezni.

Hatékonysági okokból egy adott implementáció dönthet úgy, hogy az egy partíción belül a távoli hívás interfész felé irányuló alprogramhívásokat helyi hívásként valósítja meg. Az ilyen döntés természetesen nem befolyásolhatja a program szemantikáját. Bizonyos esetekben a programozó mégis kívánatosnak tarthatja, hogy minden ilyen hívás távoli hívásként történjen, például ha azt akarja elérni, hogy minden hívás megjelenjen a partíciók közötti kommunikációs rendszeren, mondjuk a könnyebb nyomkövetés érdekében. Ilyen esetben az alábbi direktívával kényszeríthetjük erre a fordítót:

`pragma All_Calls_Remote`[(< *könyvtári egység neve* >)];

Az `All_Calls_Remote` pragma a `Remote_Call_Interface` direktívával azonos helyeken helyezhető el és csak távoli hívás interfész könyvtári egységekre vonatkozhat.

E.3. Távoli alprogramhívások

Távoli alprogramhívás esetén olyan eljárások vagy függvények végrehajtását kezdeményezhetjük amelyek egy másik partícióban helyezkednek el. Azt a partíciót, amelyben a távoli alprogramhívás szerepel, *hívó partíció*nak nevezzük, az a partíciót pedig, amely végrehajtja a hívott alprogramot, a *hívott partíció*. Távoli alprogramhívás három esetben következhet be:

[1]Mutató paraméternek olyan paramétert nevezünk, amelynek deklarációjában szerepel az `access` kulcsszó. Nem számít viszont mutató paraméternek az olyan paraméter amelynek típusa egy korábban deklarált mutató típus, így ilyen paraméter megengedett távoli hívás interfészekben deklarált alprogramoknál is.

1. Egy olyan alprogram közvetlen meghívásakor, amelyet egy távoli hívás interfészben deklaráltak.

2. Egy alprogramra vonatkozó távoli mutató típuson keresztüli indirekt hívással.

3. Egy osztályszintű típusra vonatkozó távoli mutató típus használata esetén, ha a hivatkozott operandus a dinamikus kötést vezérlő paraméter. Ilyenkor a dinamikus összekapcsolás létrejön és távoli alprogramhívás történik, ahol a hívott partíció a távoli mutató típus által hivatkozott típus definícióját tartalmazó partíció lesz.

Egy távoli hívás fordításakor a kódba egy úgynevezett *calling stub* program kerül, amely felelős a hívás létrejöttéhez szükséges kommunikáció lebonyolításáért. Híváskor ez a program elküldi a paramétereket a hívott partíció *receiving stub*-jának, amely értelmezi azokat, majd a szokásos lokális módon meghívja a megfelelő alprogramot a kapott paraméterekkel. Ha a hívott alprogram befejezte működését, a receiving stub visszaküldi a futás eredményét valamint az esetlegesen fellépett kivételeket a hívó partíciónak. A calling stub itt vagy visszaadja az eredményeket, vagy kiváltja a kapott kivételeket, amelyek így a hívó alprogramban gyűrűznek tovább. A stub-ok a kommunikációs csatornákat mint stream-eket kezelik, az adatok továbbításához a küldendő típusok `Read` és `Write` attribútumait használják.

Mint láthattuk, a távoli alprogramhívások a használat szempontjából nem különböznek a hagyományos, lokális hívásoktól. Néhány különbség azért mégis akad. Az első és legfontosabb, hogy mivel a hívás valamilyen kommunikációs csatorna igénybevételével történik, így annak hibájából meghiúsulhat a hívás. Ilyen esetben a calling stub a hívó partícióban `Communications_Error` kivételt vált ki, amelyet a `System.RPC` csomag deklarál.

Egy másik eltérés az aszinkron eljáráshívás lehetősége. Ha a hívott eljárásnak nincsenek `out` és `in out` paraméterei, akkor az alábbi direktívával előírhatjuk a fordító számára, hogy a hívás aszinkron módon történjék, vagyis a calling stub térjen vissza a hívást kezdeményező üzenet elküldése után azonnal. Így a hívó és a hívott partíciók aszinkron hajtódhatnak végre.

`pragma Asynchronous(`< *eljárás neve* >`);`

Aszinkron hívás esetén, ha a hívott eljárásban valamilyen kivétel lép fel, az nem terjed tovább a hívó alprogramban, hanem elnyelődik.

További eltérés, hogy az osztott végrehajtás miatt a távoli alprogramhívás megelőzheti a hívott alprogram kiértékelését. Ilyenkor a hívás várakozik a kiértékelés befejezéséig, majd a megszokott módon végrehajtódik, kivéve ha a hívó partíció közben érvényteleníti a hívást.

E.4. Dinamikus összekapcsolás

Az egyes aktív partíciók általában külön címtérben futnak. Ez azt jelenti, hogy ugyanaz a memóriacím más-más objektumot jelölhet két különböző partícióban. Éppen ezért egy, az egyik partícióhoz tartozó, objektum címe nem használható a másik partícióban. Mivel a mutató típusok nagyrészt a memóriacímekre épülnek, így a mutató típusú értékek átadása a partíciók között veszélyes, használata bizonyos megszorításokat igényel. Éppen ezért távoli hívás interfész csomagokban csak kétféle mutató típus deklarálható:

1. Alprogramra vonatkozó mutató típus
2. Osztályszintű, korlátozott átlátszatlan jelölt típusra vonatkozó mutató típus.

Ezen mutató típusok átadhatóak partíciók között távoli alprogramhívásokban, mivel tartalmazzák az általuk hivatkozott objektumot tartalmazó partíció azonosításához szükséges információt is. Segítségükkel kétféle dinamikus kötés valósítható meg a távoli alprogramhívások során.

E.4.1. Alprogramra vonatkozó mutató típus

Alprogramra vonatkozó távoli mutató típus csak olyan alprogramra hivatkozhat, amelyet valamely távoli hívás interfész könyvtári egységben deklaráltunk. Minden ilyen típusú érték tartalmazza a hivatkozott alprogram törzsét tartalmazó partíció azonosításához szükséges információt. Az így hivatkozott alprogram hívása mindig távoli hívást eredményez. Használatára tekintsük az alábbi példát[2], amelyben egy időzítő szervert specifikálunk. A szerver időzítési kérelmeket fogad, amelyek megadnak egy alprogramot, amelyet a nap megadott időpillanatában kell elindítani.

E.1. Példa *(Eljárásra vonatkozó távoli mutató típus)*

```
package Idôzítô_Szerver is
    pragma Remote_Call_Interface;
    type Idôzített_Esemény is access procedure;
    procedure Új_Esemény(Indítási_idô: in Duration;
                Esemény: in Idôzített_Esemény);
end Idôzítô_Szerver;
```

Tegyük fel, hogy az egyik kliens partícióban definiált egy `Fájl_Mentés` eljárás amelyet hajnali 1:00-kor kell elindítani (a nap 3600. másodpercében), illetve van egy `Projekt_Újrafordítás` nevű, amelyet pedig 2:00-kor (7200. másodperc) kell lefuttatni. A kliens az alábbi könyvtári egységeket deklarálja:

[2] Az alább következő példdák mind [6] alapján készültek.

E.2. Példa *(Távoli hívás interfész kñyvtári egységek)*

```
package Mentés is
    pragma Remote_Call_Interface;
    procedure Fájl_Mentés;
end Mentés;

package Újrafordítás is
    pragma Remote_Call_Interface;
    procedure Projekt_Újrafordítás;
end Recompilation;
```

A kliens ezután a következő módon regisztráltatja igényeit:

E.3. Példa *(Alprogramra vonatkozó távoli mutató típus használata)*

```
Idôzítô_Szerver.Új_Esemény(3600.0,
                           Fájl_Mentés'Access);
Idôzítô_Szerver.Új_Esemény(7200.0,
                           Projekt_Újrafordítás'Access);
```

Az `Access` attribútum itt egyaránt meghatározza a klienst futtató aktív partíció azonosítóját és azon belül az eljárások címét. Az időzítő szerver a megfelelő módon tárolja az átadott mutató értékeket, majd 1:00-kor elvégzi a szükséges távoli alprogramhívást, hasonlóan 2:00-kor.

E.4.2. Osztályszintű korlátozott típusra mutató típus

Osztályszintű korlátozott átlátszatlan típusra vonatkozó távoli mutató típusú értéket csak az `Access` (vagy a `Unchecked_Access`) attribútum segítségével állíthatunk elő, allokátor ilyen célra nem hívható. Az attribútum itt is visszaadja a megfelelő partícióazonosítót. Amikor a mutató érték, vagy a hivatkozott objektum átadódik egy dinamikusan kapcsolandó alprogramnak (azaz a korlátozott típus egy primitív műveletének), akkor egy üzenet generálódik a mutató értékben megjelölt partíció felé és a dinamikus összekapcsolás abban a partícióban jön létre. A megfelelő szabályok biztosítják, hogy a dinamikus összekapcsolás mindig olyan alprogramverzióhoz vezessen, amely az adott aktív partícióban vagy egy általa elérhető osztott passzív partícióban található. Amennyiben több ilyen paramétert adunk át egyetlen dinamikusan kapcsolandó hívásban akkor külön ellenőrzést végez a rendszer, hogy a mutató értékek ugyanarra a partícióra mutatnak-e. Ha ez nem teljesül, akkor `Constraint_Error` kivétel lép fel.

Tegyük fel, hogy egy elosztott program tartalmaz egy aktív partíciót, amely egyik szolgáltatása bizonyos grafikus adatok előállítását eredményezi.

Különböző kliens partíciók igényelhetik ezt a szolgáltatást, hogy annak eredményét az általuk menedzselt grafikus egységen (monitor, nyomtató, plotter stb.) megjelenítsék. Definiálunk egy absztrakt korlátozott átlátszatlan `Grafikus_Egység` típust, amelynek műveletei megfelelnek a szerver által használt grafikus műveleteknek. Ezután a szerver szolgáltató műveletét egy `Grafikus_Egység'Class` típusú objektumra vonatkozó mutató értékkel hívjuk meg, amely azonosítja azt az output perifériát amelyen az eredményt látni kívánjuk.

E.4. Példa *(Távoli típus könyvtári egység)*

```
package Grafikus_Egységek is
   pragma Remote_Types;

   type  Grafikus_Eység is abstract tagged
                           limited private;

   procedure Határoló_Doboz
             (Egység: in out Grafikus_Egység,
              Fent, Lent, Bal, Jobb: in Float);
   procedure Egyenes
             (Egység: in out Grafikus_Egység,
              Honnan_X, Honnan_Y,
              Hova_X,   Hova_Y: in Float);
   procedure Kôr
             (Egység: in out Grafikus_Egység,
              Közép_X, Közép_Y, Sugár: in Float);
   ...
   type Grafikus_Egység_Mutató is
        access all Grafikus_Egység'Class;
private
   type Grafikus_Egység is abstract tagged
                           limited null record;
end Graphical_Devices;

with Grafikus_Egységek;
package Grafikai_Szolgáltatások is
   pragma Remote_Call_Interface;
   procedure Szolgáltatás
      (...; Kimenet: in Grafikus_Egység_Mutató);
   ...
end Grafikai_Szolgáltatások;
```

Ezután a kliensek az ő saját speciális grafikus eszközükhöz úgy definiálják a meghajtót, hogy származtatnak egy konkrét típust a `Grafikus_Egység` típusból. A származtatott típus objektumai megfelelnek az azonos típusú

eszközöknek (két azonos nyomtató két önálló, ám azonos típusú objektum). Ezután egy távoli mutató érték, amely a megfelelő eszközt leíró objektumra hivatkozik, átadható a `Szolgáltatás` hívásakor, annak `Kimenet` paraméterében. Amikor a `Szolgáltatás` törzsében egy grafikus kimenetet adó hívás történik, például

```
Határoló_Doboz(Kimenet.all, 0.0, 50.0, 0.0, 50.0);
```

egy távoli alprogramhívás generálódik afelé a partíció felé, amely a `Kimenet` által hivatkozott objektumot tartalmazza és ahol a vezérlés a dinamikusan meghatározott `Határoló_Doboz` verziónak adódik.

A dinamikus távoli alprogramhívások nagy rugalmasságot eredményeznek az elosztott programok szerkezetében, mivel így a hívást tartalmazó fordítási egység nem kell, hogy szemantikusan függjön a hívott alprogramot definiáló egységtől. Így az időzítő szervert definiáló csomag egyáltalán nem függ a `Fájl_Mentés` és `Projekt_Újrafordítás` eljárásokat tartalmazó klienstől, sőt éppen ellenkezőleg, a kliens függ a `Új_Esemény` hívás miatt a szerverhez tartozó távoli hívás interfész egységtől.

Hasonlóan, a grafikus kimenetet adó szerver is olyan `Grafikus_Egység` típusú egységekkel végez műveleteket amelyek definícióját tartalmazó egységektől szemantikusan nem függ. Ennek következtében, mialatt a szerver partíciója folyamatosan működik, a kliens partíciók, amelyek az egyes konkrét grafikus egységek működését felügyelik, szabadon ki-bekapcsolhatóak. A dinamikus távoli alprogramhívás tehát lehetővé teszi a hívott alprogramok cseréjét egy működő elosztott programban. Lehetőségünk nyílik tehát olyan non-stop programok készítésére amelynek partíciói menet közben cserélhetőek, karbantarthatóak, ki- és bekapcsolhatóak.

F. Függelék

Információs rendszerek

Ez a függelék az alábbi – az információs rendszerek programozásához kapcsolódó – eszközöket tárgyalja:

- egy decimális altípus gépi alapszámának definiálása,
- a `Decimal` csomag, amely a decimális típusok ábrázolására vonatkozó implementációs korlátokat meghatározó konstansokat és egy decimális osztást végző sablon eljárást tartalmaz,
- a `TextIO.Editing` és `Wide_Text_IO.Editing` csomagok, amelyek decimális adatok formázott megjelenítését támogatják.

F.1. A gépi alapszám definiálása

Egy decimális altípus gépi ábrázolásának alapszámát a `Machine_Radix` attribútum határozza meg. Értékét megadhatjuk egy attribútum-definíciós utasítással, amelyben egy olyan statikus kifejezés szerepelhet, melynek értéke 2 vagy 10. Az alapszám megadását egy példán szemléltetjük:

```
type Money is delta 0.01 digits 15;
for Money'Machine_Radix use 10;
```

F.2. A `Decimal` csomag

A `Decimal` könyvtári csomag deklarációja:

```
package Ada.Decimal is
   pragma Pure(Decimal);

   Max_Scale : constant := implementation-defined;
   Min_Scale : constant := implementation-defined;
```

```
   Min_Delta : constant := 10.0**(-Max_Scale);
   Max_Delta : constant := 10.0**(-Min_Scale);

   Max_Decimal_Digits : constant := implementation-defined;
```

Max_Scale a legnagyobb n egész szám, amelyre 10^{-n} egy decimális típus definíciójában delta értékként szerepelhet. Típusa univerzális egész.

Hasonlóan Min_Scale a legkisebb n egész szám, amelyre 10^{-n} egy decimális típus definíciójában delta értékként szerepelhet. Típusa univerzális egész.

Min_Delta egy decimális fixpontos típus definíciójában megengedett legkisebb delta érték. Típusa univerzális valós.

Hasonlóan Max_Delta egy decimális fixpontos típus definíciójában megengedett legnagyobb delta érték. Típusa univerzális valós.

Max_Decimal_Digits a legnagyobb egész szám, amely egy decimális fixpontos típus definíciójában digits értékként szerepelhet. Típusa univerzális egész.

```
   generic
      type Dividend_Type  is delta <> digits <>;
      type Divisor_Type   is delta <> digits <>;
      type Quotient_Type  is delta <> digits <>;
      type Remainder_Type is delta <> digits <>;
   procedure Divide (Dividend  : in Dividend_Type;
                     Divisor   : in Divisor_Type;
                     Quotient  : out Quotient_Type;
                     Remainder : out Remainder_Type);
   pragma Convention(Intrinsic, Divide);

end Ada.Decimal;
```

A Divide művelet az out paramétereit az alábbiak szerint állítja be: Quotient értéke Quotient_Type(Dividend/Divisor), Remainder értéke pedig Remainder_Type(I), ahol I megegyezik a Dividend paraméter és a Divisor*Quotient szorzat közötti eltéréssel.

F.3. Decimális típusok formázott megjelenítése

A Text_IO.Editing és Wide_Text_IO.Editing csomagok felhasználásával megvalósítható a decimális típusú értékek lokalizált megjelenítése. A megjelenítendő sztring három dologtól függ: a megjelenítendő numerikus értéktől,

a programban specifikálható lokál-elemektől, és egy formázásvezérlő értéktől. A lokál-elemek:

- valuta sztring
- a számjegy-csoport elhatároló karakter („ezredes vessző")
- az alappontot jelző karakter („tizedes pont")
- a numerikus értékek bevezető nulláinak helyettesítésére szolgáló karakter

A `Text_IO.Editing` csomag esetében a megjelenített szöveg és a valuta sztring `String` típusú, és a lokál-karakterek `Character` típusúak, míg a `Wide_Text_IO.Editing` csomagban a típusok értelemszerűen `Wide_String` és `Wide_Character`. Mindegyik lokál elemnek van alapértelmezése, amely kicserélhető, vagy explicit felülbírálható.

A formázást a `Picture` típus egy értéke vezérli. A `Picture` egy átlátszatlan típus, amely meghatározza a formázott sztring összetételét: vezérli az előjel formáját és elhelyezését, a lokál-elemek és a számjegyek pozícióját, az alappont meglétét vagy hiányát, a bevezető nullák elnyomását és bizonyos karakterek beszúrását.

Egy `Picture` objektum egy picture sztringnek nevezett sztringből és egy logikai értékből áll. A picture sztring a megjelenítendő sztringnek egy mintája, a logikai érték pedig azt határozza meg, hogy a nulla értékből egy csupa szóközből álló sztringet kell-e létrehozni. A picture sztring egy- vagy kétkarakteres szimbólumok sorozata, amelyből a formázott sztringet úgy kapjuk, hogy minden szimbólum helyére egy karaktert vagy sztringet helyettesítünk a megfelelő pozíción. A szimbólumokat a hatásuk alapján kategorizáljuk:

Számjegy:	`'9'`
Alappont vezérlés:	`'.'` `'V'`
Előjel vezérlés:	`'+'` `'-'` `'<'` `'>'` `"CR"` `"DB"`
Valutanem vezérlés:	`'$'` `'#'`
Nulla elnyomás:	`'Z'` `'*'`
Beszúrás:	`'_'` `'B'` `'0'` `'/'`

A fenti szimbólumokban szereplő betűk előfordulhatnak kisbetűs formában is, a hatásuk ekkor is ugyanaz marad.

A picture sztringben egy `'9'` karakter előfordulása egy decimális számjegy helyét jelöli a formázott sztring azon pozícióján.

Egy alappont vezérlő karakter előfordulása az alappont helyét jelöli ki a formázott sztringben. A `'.'` karakter az aktuális karakter pozíciót, míg a `'V'` karakter a feltételezett karakter pozíciót jelzi.

Az előjel vezérlő karakternek az előjel megjelenítésére van hatása. A '<' és '>' karakter értékek hatására a negatív értékek zárójelbe kerülnek. Ha a '+', '-' vagy '<' karakterek egyszeresen fordulnak elő a picture sztringben, akkor az egy rögzített pozíciót jelöl ki az előjel számára, míg ha többször ismétlődnek, akkor az egy lebegő előjel-pozíciót jelez, azaz az előjelet a formázott sztringben megelőzik a bevezető nullákat helyettesítő szóközök.

Egy valutanem vezérlő karakter előfordulása a picture sztringben a valutanem sztring beszúrását eredményezi a formázott sztringben. A '$' karakter a teljes valutanem sztringet jelzi, míg a '#' karakter a valutanem sztring egy karakterét jelzi. A '$' karakter egyszeres és többszörös előfordulására ugyanaz a szabály vonatkozik, mint amit az előjelek esetén láthattunk. A '#' karakter egyszeres vagy többszörös előfordulása a környezettől függően jelezhet rögzített vagy lebegő-pozíciós megjelenítést is.

Egy nulla elnyomó karakter a picture sztringben azt jelzi, hogy a bevezető nulla számjegyeket szóközre ('Z' esetén), vagy kitöltő karakterre ('*' esetén) kell cserélni.

A '0' és '/' beszúró karakterek helyére önmaguk, a 'B' karakter helyére szóköz, a '_' karakter helyére pedig a csoport-elválasztó karakter kerül a formázott sztringben.

Példaként tekintsük az alábbi picture sztringet: "<Z_ZZ9.99##>", és tegyük fel, hogy a valutanem sztring "Ft", a csoport-elválasztó karakter '.' és az alappont ','. Ekkor a 32.10 és -5432.10 decimális értékek formázott alakjai: " 32,10Ft ", illetve "(5.432,10Ft)" lesz.

Azt mondjuk, hogy egy picture sztring jól formált, ha eleget tesz bizonyos szintaktikai szabályoknak, kompozíciós megszorításoknak és karakter ismétlési konvencióknak. Ezen feltételrendszer részletes leírása megtalálható a nyelv referencia kézikönyvében, itt nem részletezzük. Helyette álljon itt néhány példa, amely tükrözi a picture sztring használatát:

Érték	Picture	Lokál	Formázott sztring
123456.78	"-###**_***_**9.99"		" $***123,456.78"
		(1)	" FF***123.456,78"
123456.78	"-$$$**_***_**9.99"		" $***123,456.78"
		(1)	" FF***123.456,78"
0.0	"-$$$$$$.$$"		" "
0.20	"-$$$$$$.$$"		" $.20"
-1234.565	"<<<<_<<<.<<###>"	(2)	" (1,234.57DM)"
12345.67	"###_###_##9.99"	(3)	" CHF12,345.67"

A fenti táblázatban (1)-gyel jelöltük azt a lokált, melyben a valutanem sztring "FF", a csoport-elválasztó karakter '.' és az alappont karakter ','. A (2)-vel jelölt lokálban a valutanem sztring "DM", míg a (3)-mal jelöltben "CHF".

F.4. A Text_IO.Editing csomag

A Text_IO.Editing csomag a Picture átlátszatlan típust és műveleteit, valamint a Decimal_Output sablon csomagot tartalmazza. A Picture típus egy példánya egy jólformált picture sztringből és egy logikai értékből áll, amely azt mutatja, hogy a nulla numerikus értéket – a picture sztring tartalmától függetlenül – egy csak szóközökből álló sztringként kell-e megjeleníteni. A Decimal_Output csomag az előbbiekben vázlatosan ismertetett transzformációkat elvégző eljárásokat tartalmazza. A Text_IO.Editing csomag deklarációja:

```
package Ada.Text_IO.Editing is

   type Picture is private;

   function Valid (Pic_String      : in String;
                   Blank_When_Zero : in Boolean := False)
      return Boolean;
```

A Valid függvény akkor ad vissza True értéket, ha Pic_String jólformált picture sztring, amelynek hossza nem nagyobb mint Max_Picture_Length és vagy Blank_When_Zero értéke False, vagy Pic_String nem tartalmaz '*' karaktert.

```
   function To_Picture (Pic_String      : in String;
                        Blank_When_Zero : in Boolean := False)
      return Picture;
```

A To_Picture függvény egy Picture objektumot konstruál a megadott picture sztringből és logikai értékből. Picture_Error hibajelzést kapunk, ha Valid(Pic_String, Blank_When_Zero) nem teljesül.

```
   function Pic_String      (Pic : in Picture)
      return String;
   function Blank_When_Zero (Pic : in Picture)
      return Boolean;
```

Legyen Pic a To_Picture(String_Item, Boolean_Item) függvény által visszaadott érték. Ekkor Pic_String(Pic) értéke egy a String_Item-mel ekvivalens kiterjesztett picture sztring, melyben minden előforduló betű nagybetű, Blank_When_Zero(Pic) értéke pedig Boolean_Item.

```
   Max_Picture_Length  : constant := implementation_defined;
   Picture_Error       : exception;
```

A `Max_Picture_Length` konstans a picture sztringek maximális megengedett hosszát tartalmazza. Ez az érték a nyelv egyes implementációiban más és más lehet. A `Picture_Error` kivételt a `To_Picture` művelet válthatja ki.

```
Default_Currency    : constant String    := "$";
Default_Fill        : constant Character := '*';
Default_Separator   : constant Character := ',';
Default_Radix_Mark  : constant Character := '.';
```

A fenti négy konstans a lokál egyes elemeinek alapértelmezését definiálja.

```
generic
   type Num is delta <> digits <>;
   Default_Currency   : in String    :=
      Text_IO.Editing.Default_Currency;
   Default_Fill       : in Character :=
      Text_IO.Editing.Default_Fill;
   Default_Separator  : in Character :=
      Text_IO.Editing.Default_Separator;
   Default_Radix_Mark : in Character :=
      Text_IO.Editing.Default_Radix_Mark;
package Decimal_Output is
```

A `Decimal_Output` sablon csomag a numerikus típussal, valamint a lokál elemeivel paraméterezhető.

```
   function Length (Pic      : in Picture;
                    Currency : in String
                       := Default_Currency)
      return Natural;
```

A `Length` függvény visszatérési értéke `Pic_String(Pic)'Length + V - P`, ahol `V = Currency'Length-1`, ha `Pic` tartalmaz `'\$'` karaktert, és 0 különben, illetve `P = 1`, ha `Pic` tartalmaz `'V'` vagy `'v'` karaktert, és 0 különben.

```
   function Valid (Item     : in Num;
                   Pic      : in Picture;
                   Currency : in String
                      := Default_Currency)
      return Boolean;
```

Ha `Image(Item, Pic, Currency)` nem vált ki `Layout_Error` kivételt, akkor a `Valid` függvény `True` értéket ad vissza, ellenkező esetben a visszatérési érték `False`.

```
function Image (Item       : in Num;
                Pic        : in Picture;
                Currency   : in String
                   := Default_Currency;
                Fill       : in Character
                   := Default_Fill;
                Separator  : in Character
                   := Default_Separator;
                Radix_Mark : in Character
                   := Default_Radix_Mark)
   return String;
```

Az `Image` függvény a paramétereinek megfelelő formázott sztringet adja vissza. Ha a fenti paraméterek nem felelnek meg a formázási szabályoknak, akkor `Layout_Error` hibajelzést kapunk.

```
procedure Put (File       : in File_Type;
               Item       : in Num;
               Pic        : in Picture;
               Currency   : in String
                  := Default_Currency;
               Fill       : in Character
                  := Default_Fill;
               Separator  : in Character
                  := Default_Separator;
               Radix_Mark : in Character
                  := Default_Radix_Mark);

procedure Put (Item       : in Num;
               Pic        : in Picture;
               Currency   : in String
                  := Default_Currency;
               Fill       : in Character
                  := Default_Fill;
               Separator  : in Character
                  := Default_Separator;
               Radix_Mark : in Character
                  := Default_Radix_Mark);
```

Mindkét `Put` eljárás az `Image(Item, Pic, Currency, Fill, Separator, Radix_Mark)` sztringet írja ki, a `Text_IO.Put` műveletekkel analóg módon.

```
procedure Put (To         : out String;
               Item       : in Num;
```

```
                   Pic        : in Picture;
                   Currency   : in String
                      := Default_Currency;
                   Fill       : in Character
                      := Default_Fill;
                   Separator  : in Character
                      := Default_Separator;
                   Radix_Mark : in Character
                      := Default_Radix_Mark);
```

A fenti Put művelet az Image(Item, Pic, Currency, Fill, Separator, Radix_Mark) sztringet jobbra igazítva a To paraméterbe másolja. A To paraméterben így ki nem töltött karakterek értéke szóköz lesz. Ha To nem elég hosszú a formázott sztring befogadásához, akkor Layout_Error hibajelzést kapunk.

```
   end Decimal_Output;
private
   ...
end Ada.Text_IO.Editing;
```

Megjegyezzük még, hogy mind az Image függvény, mind a Put eljárások Constraint_Error kivételt váltanak ki, ha Currency értéke null sztring.

F.5. A Wide_Text_IO.Editing csomag

A Wide_Text_IO.Editing csomag tartalma és szemantikája megegyezik a Text_IO.Editing csomagéval, az alábbi változtatásokkal:

- mindenhol, ahol Character típus szerepelt, itt Wide_Character típus szerepel,
- a Text_IO azonosító minden előfordulása Wide_Text_IO-ra cserélődik,
- a Default_Currency konstans típusa String, helyett Wide_String
- a Decimal_Output csomagban a String típus minden előfordulása helyett Wide_String szerepel.

Természetesen a picture sztringre vonatkozó műveletekben a paraméter String marad, hiszen a picture sztring felépítése nem változik, rá ugyanazok a szabályok vonatkoznak mint a Text_IO.Editing csomag esetén.

G. Függelék

Numerikus műveletek

Ebben a függelékben az Ada és a számábrázolás kapcsolatát vizsgáljuk, kitérve az alábbi részterületekre:

- komplex aritmetika és input/output,
- a „szigorú numerikus mód”, amelyben a fix- és lebegőpontos számok előredefiniált műveleteinek, illetve a rájuk vonatkozó egyéb előredefiniált csomagoknak bizonyos meghatározott pontossági feltételeknek kell eleget tenniük,
- a „laza numerikus mód”, amelyben semmilyen megszorítás nincs az előredefiniált numerikus műveletek és csomagok pontosságára,
- lebegőpontos aritmetikának olyan modelljei, amelyek a szigorú mód követelményeinek alapját képezik,
- a lebegőpontos típusok szigorú módban megkövetelt modell-orientált attribútumai.

G.1. Komplex aritmetika

A komplex számok kezelésére vonatkozó típusok és matematikai műveletek definícióját a `Generic_Complex_Types` csomag tartalmazza. A matematikából ismert (komplex) elemi függvények megvalósítását (közelítő függvényérték kiszámítását) a `Generic_Complex_Elementary_Functions` csomag tartalmazza. Mindkét csomag a `Numerics` előredefiniált csomag sablon gyereke. Az előredefiniált lebegőpontos típusokra vannak ezeknek a csomagoknak nem-sablon változatai, amelyek szintén a `Numerics` csomag gyerekei.

G.1.1. Komplex típusok

Az Ada nyelv `Numerics.Generic_Complex_Types` sablon csomagot az alábbi módon deklarálja:

```
generic
   type Real is digits <>;
package Ada.Numerics.Generic_Complex_Types is
   pragma Pure(Generic_Complex_Types);
   type Complex is
      record
         Re, Im : Real'Base;
      end record;
```

A `Complex` típus egy látható típus, amely a komplex számot algebrai alakban (derékszögű koordinátarendszerbeli koordinátákkal) ábrázolja. A nyelv azért használja az algebrai ábrázolást, mert a gyakorlatban sokkal gyakoribb, mint a polárkoordinátás ábrázolás. A komplex műveletek és elemi függvények pontossága reprezentációfüggő, ezért az implementálóknak ismerniük kell a reprezentációt. A típus azért látható, hogy lehetőség legyen összeg-literálok használatára.

```
   type Imaginary is private;

   i : constant Imaginary;
   j : constant Imaginary;
```

Az `Imaginary` típus és az `i`, illetve `j` konstansoknak az a célja, hogy írhassunk `(a+b*i)` alakú literálokat.

```
   function Re (X : Complex)   return Real'Base;
   function Im (X : Complex)   return Real'Base;
   function Im (X : Imaginary) return Real'Base;
```

Az `Re` és `Im` függvények jelentése megegyezik a matematikában szokásossal.

```
   procedure Set_Re (X  : in out Complex;
                     Re : in     Real'Base);
   procedure Set_Im (X  : in out Complex;
                     Im : in     Real'Base);
   procedure Set_Im (X  :    out Imaginary;
                     Im : in     Real'Base);
```

A `Set_Re` és `Set_Im` eljárások a megadott komplex paraméter megfelelő komponensét kicserélik a megadott értékre. Tisztán képzetes (`Imaginary`) típusú értékre alkalmazva a megadott valós értéknek megfelelő képzetes értéket adja vissza.

```
function Compose_From_Cartesian (Re, Im : Real'Base)
   return Complex;
function Compose_From_Cartesian (Re     : Real'Base)
   return Complex;
function Compose_From_Cartesian (Im     : Imaginary)
   return Complex;
```

A Compose_From_Cartesian függvények komplex értéket képeznek a megadott paraméterekből. Ott, ahol csak az egyik komponenst adjuk meg, a másik automatikusan 0 lesz.

```
function Modulus (X     : Complex) return Real'Base;
function "abs"   (Right : Complex) return Real'Base
   renames Modulus;

function Argument (X     : Complex)   return Real'Base;
function Argument (X     : Complex;
                   Cycle : Real'Base) return Real'Base;
```

A Modulus és az Argument függvények a komplex szám polárkoordinátáit adják vissza (a Modulus a hossz, az Argument pedig a szög). Ennek megfelelően a Modulus függvény értéke nemnegatív. Az Argument függvénynek az a változata, amelyben szerepel a Cycle paraméter, a szöget olyan mértékegységben adja meg, amelyből a teljes kör pontosan a Cycle paraméter értéke. A Cycle paraméter nélküli Argument függvény a szöget radiánban adja meg. A függvény értéke a -Pi..Pi (illetve a -Cycle/2..Cycle/2) intervallumba esik. Ha a komplex paraméter képzetes része 0 és Real'Signed_Zeros értéke True, akkor a függvényérték előjele megegyezik a képzetes rész előjelével. Ha Cycle értéke nem pozitív, akkor Numerics.Argument_Error hibajelzést kapunk.

```
function Compose_From_Polar(Modulus, Argument: Real'Base)
   return Complex;
function Compose_From_Polar(Modulus, Argument,
   Cycle : Real'Base) return Complex;
```

A Compose_From_Polar függvények egy komplex számot állítanak elő a polárkoordináták alapján az alábbi szabályok szerint: a visszaadott komplex szám komponensei annak a pontnak a koordinátái lesznek, amely az origótól a Modulus paraméter abszolút értékével megegyező távolságra van, és pozitív Modulus esetén a pozitív, negatív Modulus esetén pedig a negatív valós féltengelytől az óramutató járásával ellentétes irányban az Argument paraméterben megadott szög alatt látszik. A Cycle paraméter jelentése megegyezik az Argument függvénynél megismerttel. Numerics.Argument_Error hibajelzést kapunk, ha Cycle értéke nem pozitív.

```
function "+"       (Right : Complex) return Complex;
function "-"       (Right : Complex) return Complex;
function Conjugate (X     : Complex) return Complex;

function "+" (Left, Right : Complex) return Complex;
function "-" (Left, Right : Complex) return Complex;
function "*" (Left, Right : Complex) return Complex;
function "/" (Left, Right : Complex) return Complex;

function "**" (Left : Complex; Right : Integer)
   return Complex;

function "+"       (Right : Imaginary) return Imaginary;
function "-"       (Right : Imaginary) return Imaginary;
function Conjugate (X     : Imaginary) return Imaginary
   renames "-";
function "abs"     (Right : Imaginary) return Real'Base;

function "+" (Left, Right : Imaginary) return Imaginary;
function "-" (Left, Right : Imaginary) return Imaginary;
function "*" (Left, Right : Imaginary) return Real'Base;
function "/" (Left, Right : Imaginary) return Real'Base;

function "**" (Left : Imaginary; Right : Integer)
   return Complex;

function "<"  (Left, Right : Imaginary) return Boolean;
function "<=" (Left, Right : Imaginary) return Boolean;
function ">"  (Left, Right : Imaginary) return Boolean;
function ">=" (Left, Right : Imaginary) return Boolean;

function "+" (Left : Complex;   Right : Real'Base)
   return Complex;
function "+" (Left : Real'Base; Right : Complex)
   return Complex;
function "-" (Left : Complex;   Right : Real'Base)
   return Complex;
function "-" (Left : Real'Base; Right : Complex)
   return Complex;
function "*" (Left : Complex;   Right : Real'Base)
   return Complex;
function "*" (Left : Real'Base; Right : Complex)
   return Complex;
function "/" (Left : Complex;   Right : Real'Base)
```

```
      return Complex;
   function "/" (Left : Real'Base; Right : Complex)
      return Complex;

   function "+" (Left : Complex;   Right : Imaginary)
      return Complex;
   function "+" (Left : Imaginary; Right : Complex)
      return Complex;
   function "-" (Left : Complex;   Right : Imaginary)
      return Complex;
   function "-" (Left : Imaginary; Right : Complex)
      return Complex;
   function "*" (Left : Complex;   Right : Imaginary)
      return Complex;
   function "*" (Left : Imaginary; Right : Complex)
      return Complex;
   function "/" (Left : Complex;   Right : Imaginary)
      return Complex;
   function "/" (Left : Imaginary; Right : Complex)
      return Complex;

   function "+" (Left : Imaginary; Right : Real'Base)
      return Complex;
   function "+" (Left : Real'Base; Right : Imaginary)
      return Complex;
   function "-" (Left : Imaginary; Right : Real'Base)
      return Complex;
   function "-" (Left : Real'Base; Right : Imaginary)
      return Complex;
   function "*" (Left : Imaginary; Right : Real'Base)
      return Imaginary;
   function "*" (Left : Real'Base; Right : Imaginary)
      return Imaginary;
   function "/" (Left : Imaginary; Right : Real'Base)
      return Imaginary;
   function "/" (Left : Real'Base; Right : Imaginary)
      return Imaginary;

private
   type Imaginary is new Real'Base;
   i : constant Imaginary := 1.0;
   j : constant Imaginary := 1.0;

end Ada.Numerics.Generic_Complex_Types;
```

A `Conjugate` függvény a komplex szám konjugáltját adja vissza, az aritmetikai műveletek jelentése pedig megegyezik a matematikában megszokottal.

Ha `Real'Machine_Overflows` értéke `True`, akkor az osztás operátor 0 osztó érték esetén, a hatványozás operátor pedig 0 alap és negatív kitevő esetén `Constraint_Error` kivételt vált ki. Ha `Real'Machine_Overflows` értéke `False`, akkor a fenti argumentumok esetén a visszatérési érték nemdefiniált.

G.1.2. Komplex elemi függvények

A `Numerics.Generic_Complex_Elementary_Functions` csomag az elemi függvényeket valósítja meg. A csomag deklarációja az alábbi:

```
with Ada.Numerics.Generic_Complex_Types;
generic
   with package Complex_Types is
      new Ada.Numerics.Generic_Complex_Types (<>);
   use Complex_Types;
package Ada.Numerics.Generic_Complex_Elementary_Functions is
   pragma Pure(Generic_Complex_Elementary_Functions);

   function Sqrt (X : Complex)   return Complex;
   function Log  (X : Complex)   return Complex;
   function Exp  (X : Complex)   return Complex;
   function Exp  (X : Imaginary) return Complex;
   function "**" (Left : Complex;   Right : Complex)
      return Complex;
   function "**" (Left : Complex;   Right : Real'Base)
      return Complex;
   function "**" (Left : Real'Base; Right : Complex)
      return Complex;

   function Sin (X : Complex) return Complex;
   function Cos (X : Complex) return Complex;
   function Tan (X : Complex) return Complex;
   function Cot (X : Complex) return Complex;

   function Arcsin (X : Complex) return Complex;
   function Arccos (X : Complex) return Complex;
   function Arctan (X : Complex) return Complex;
   function Arccot (X : Complex) return Complex;

   function Sinh (X : Complex) return Complex;
```

```
    function Cosh (X : Complex) return Complex;
    function Tanh (X : Complex) return Complex;
    function Coth (X : Complex) return Complex;

    function Arcsinh (X : Complex) return Complex;
    function Arccosh (X : Complex) return Complex;
    function Arctanh (X : Complex) return Complex;
    function Arccoth (X : Complex) return Complex;

end Ada.Numerics.Generic_Complex_Elementary_Functions;
```

Az `Ada.Numerics.Complex_Elementary_Functions` könyvtári csomag pontosan ugyanazokat a függvényeket definiálja mint a fenti sablon csomag, azzal a különbséggel, hogy a `Real'Base` típus helyett `Float`, a `Complex` és `Imaginary` típus helyett pedig a `Numerics.Complex_Types` csomag által exportált `Complex` és `Imaginary` típusok szerepelnek.

Az `Exp` függvény tisztán képzetes argumentumú változata egy alternatív lehetőséget ad a felhasználónak arra, hogy polárkoordinátákkal megadott értékből komplex számot állítson elő: a `Compose_From_Polar(Rho, Theta)` hívás mellett használható a `Rho * Exp(i * Theta)` kifejezés is.

A függvények a matematikából megismert megfelelőik közelítő értékét számolják ki. A fenti függvények azonban bizonyos körülmények között nem folytonosak:

- Az `Sqrt` és `Log` függvények értékének szakadása van, ha az `X` paraméter keresztezi a negatív valós tengelyt.
- A komplex alapú hatványozás operátor értékének szakadása van, ha az alap keresztezi a negatív valós tengelyt.
- Az `Arcsin` és `Arccos` függvény értéke valós részének, illetve az `Arctanh` függvény értéke képzetes részének szakadása van, amint az `X` paraméter metszi a valós tengely `-1.0`-tól balra vagy `1.0`-tól jobbra eső részét.
- Az `Arctan` függvény értéke valós részének, illetve az `Arcsinh` függvény értéke képzetes részének szakadása van, amint az `X` értéke metszi a képzetes tengelyt a `-i` érték alatt vagy az `i` érték felett.
- Az `Arccot` függvény értéke valós részének szakadása van, ha az `X` paraméter a `-i` és `i` értékek között keresztezi a képzetes tengelyt.
- Az `Arccosh` függvény értéke képzetes részének szakadása van, ha az `X` paraméter az `1.0` értéktől balra metszi a képzetes tengelyt.
- Az `Arccoth` függvény értéke képzetes részének szakadása van, ahogy az `X` paraméter keresztezi a valós tengelyt a `-1.0` és az `1.0` érték között.

Azon függvények számított értéke, amelyek matematikailag több értéket is felvehetnek, az alábbiak szerint alakul:

- Az `Sqrt` és `Arccosh` függvények értékének valós része nemnegatív.
- A `Log` függvény értékének képzetes részére ugyanaz a konvenció vonatkozik mint a `Numerics.Generic_Complex_Types` csomag természetes ciklusú `Argument` függvényére.
- Az `Arcsin` és `Arctan` függvények valós részének, valamint az `Arcsinh` és `Arctanh` függvények képzetes részének értékkészlete közelítőleg a `-Pi/2.0` és `Pi/2.0` intervallum.
- Az `Arccos` és `Arccot` függvények értékének valós része, illetve az `Arccoth` függvény értékének képzetes része beleesik a `0.0` és `Pi` által meghatározott intervallumba.
- Az `Arccosh` függvény képzetes részének értékkészlete közelítőleg a `-Pi` és `Pi` intervallum.

A csomag függvényeinek végrehajtása során az alábbi hibajelzéseket kaphatjuk:

- a hatványozás operátor `Numerics.Argument_Error` kivételt vált ki, ha az alap nulla és a kitevő valós része 0,
- a `Log`, `Cot` és `Coth` függvények `Constraint_Error` kivételt váltanak ki, ha az `X` paraméter értéke nulla,
- a hatványozás operátor `Constraint_Error` kivételt vált ki, ha az alap nulla és a kitevő valós része negatív,
- az `Arctan` és `Arccot` függvények `Constraint_Error` kivételt váltanak ki, ha az `X` paraméter `i` vagy `-i`,
- az `Arctanh` és `Arccoth` függvények `Constraint_Error` kivételt váltanak ki, ha az `X` paraméter `1.0` vagy `-1.0`.

`Constraint_Error` hibajelzést kaphatunk akkor is, ha valamely függvény véges értéke túlcsordul. Ez általában akkor fordulhat elő, ha a paraméter értéke eléggé megközelíti a függvény (valamely) pólusát.

Ha `Complex_Types.Real'Machine_Overflows` értéke `False`, akkor a pólusoknál a függvényérték nem definiált.

G.1.3. Komplex input-output

A komplex értékek formázott input-outputjára szolgáló eljárásokat a sablon `Text_IO.Complex_IO` csomag definiálja. A sablon paramétere egy olyan csomag, amely a `Numerics.Generic_Complex_Types` csomagból lett példányosítva valamilyen lebegőpontos altípusra. A csomag deklarációja az alábbi:

```
with Ada.Numerics.Generic_Complex_Types;
generic
   with package Complex_Types is
      new Ada.Numerics.Generic_Complex_Types (<>);
package Ada.Text_IO.Complex_IO is

   use Complex_Types;

   Default_Fore : Field := 2;
   Default_Aft  : Field := Real'Digits - 1;
   Default_Exp  : Field := 3;

   procedure Get (File  : in  File_Type;
                  Item  : out Complex;
                  Width : in  Field := 0);
   procedure Get (Item  : out Complex;
                  Width : in  Field := 0);
```

A `Get` eljárások számára az input sorozat valós literálok egy párja, amely a valós és a képzetes részt reprezentálja. A pár komponensei között lehet vessző és a pár köré írhatunk zárójeleket. Szóközök az egyes komponensek, az opcionális vessző és zárójelek között tetszőleges számban előfordulhatnak. Ha a `Width` paraméter értéke nulla, akkor:

- sor és oldal terminátor karakterek is előfordulhatnak ott ahol a szóközök megengedettek,
- ha nincs elválasztó vessző, akkor a két számot legalább egy szóköz vagy sorvégjel kell, hogy elhatárolja egymástól,
- ha volt nyitózárójel, akkor a beolvasás a csukózárójel beolvasásával fejeződik be, ellenkező esetben a képzetes rész beolvasása után.

Ha a `Width` paraméter értéke nem nulla, akkor:

- ha nincs elválasztó vessző, akkor a két számot legalább egy szóköz kell, hogy elhatárolja egymástól,

- `Width` számú karakter olvasódik be (hacsak nem olvasunk előbb egy sorvégjelet, amelynek hatására az olvasás befejeződik). A szóköz karakterek is beleszámítanak a beolvasott karakterek számába.

Az `Item` paraméterben az input sorozatnak megfelelő `Complex` típusú értéket kapjuk vissza. Az eljárások `Text_IO.Data_Error` kivételt váltanak ki, ha az input sorozat nem felel meg a kívánt szintaxisnak, vagy ha a beolvasott komplex érték komponensei nem esnek bele a `Complex_Types.Real` altípusba.

```
procedure Put (File : in File_Type;
               Item : in Complex;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);
procedure Put (Item : in Complex;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);
```

A `Put` eljárások az `Item` paraméter értékét mint valós számpárt írják ki az alábbi módon:

- kiírnak egy nyitózárójelet,
- kiírják `Item` valós részét olyan formában, mint a `Text_IO.Float_IO` csomag `Complex_Types.Real` típusra példányosított változatában a megfelelő `Put` művelet tenné a megadott `Fore`, `Aft` és `Exp` értékekkel,
- kiírnak egy vesszőt,
- kiírják az `Item` paraméter képzetes részét ugyanolyan formában, mint a `Text_IO.Float_IO` csomag `Complex_Types.Real` típusra példányosított változatának megfelelő Put művelete tenné a megadott `Fore`, `Aft` és `Exp` értékekkel,
- kiírnak egy csukózárójelet.

Ha a fájlban korlátozva van a sorhossz, akkor bármelyik fenti egység előtt opcionálisan kiírásra kerülhet egy sorvégjel.

```
procedure Get (From : in  String;
               Item : out Complex;
               Last : out Positive);
```

A fenti `Get` művelet a megadott sztring elejéről beolvas egy komplex értéket. A sztring tartalmára ugyanazok a szabályok vonatkoznak, mint a fájlból olvasó `Get` műveletnél az input sorozatra, azzal az eltéréssel, hogy a sztring végét sorvégjelnek tekintjük. Az `Item` paraméterben az input sorozatnak megfelelő `Complex` típusú értéket kapjuk vissza. A `Last` paraméterben az utolsó beolvasott karakter indexét adja vissza. Ha az input sorozat nem felel meg a kívánt szintaxisnak, vagy ha a beolvasott komplex érték komponensei nem esnek bele a `Complex_Types.Real` altípusba, akkor az eljárás `Text_IO.Data_Error` kivételt vált ki.

```
   procedure Put (To   : out String;
                  Item : in  Complex;
                  Aft  : in  Field := Default_Aft;
                  Exp  : in  Field := Default_Exp);
end Ada.Text_IO.Complex_IO;
```

Ez a `Put` eljárás az `Item` paraméter értékét valós számpárként kiírja `To` sztringbe, az alábbi szabályok alapján:

- egy nyitózárójel, a valós komponens és egy vessző balra igazítva kerül a sztringbe, ahol a valós rész formátuma megegyezik a fájlba író `Put` műveletnél meghatározottal úgy, hogy a `Fore` paraméter értékét 0-nak vesszük,
- a képzetes rész és egy csukózárójel jobbra igazítva kerül a sztringbe, ahol a képzetes rész formátuma megegyezik a fájlba író `Put` műveletnél meghatározottal úgy, hogy a `Fore` paraméter értékét akkorának választjuk, hogy az kitöltse az output sztring közepét.

Az eljárás `Text_IO.Layout_Error` kivételt vált ki, ha az output sztring túl rövid a formázott karaktersorozat befogadására.

G.1.4. A `Wide_Text_IO.Complex_IO` csomag

A `Wide_Text_IO.Complex_IO` csomag is a nyelv szabványos könyvtárai közé tartozik. A csomag deklarációja megegyezik a `Text_IO.Complex_IO` csomagéval úgy, hogy a `String` paraméter helyett mindenütt `Wide_String` szerepel. A csomag szemantikája is megegyezik az előzőekben leírtakkal, azzal a különbséggel, hogy a speciális karakterek helyett `Wide_Character` típusbeli megfelelőik fordulnak elő.

G.2. Hatékonysági követelmények

A nyelv egyes implementációinak biztosítaniuk kell egy felhasználó által választható módot, amely bizonyos – a nyelv specifikációjában leírt – pontossági és hatékonysági követelményeknek megfelel. Ez a mód hatással van

a valós típusok előredefiniált műveleteinek eredményére, a Numerics csomag leszármazottaiban található eljárások működésére, és indirekt módon a többi szabványos csomag műveleteire is. Ezt a módot *szigorú numerikus módnak* nevezzük.

Ezzel szemben kell biztosítani egy olyan – *laza numerikus módnak* nevezett – módot is, amelyre ez a követelményrendszer nem feltétlenül teljesül. A két mód közül bármelyik lehet az alapértelmezett, sőt az sem követelmény, hogy a két mód különbözzön.

G.2.1. A lebegőpontos aritmetika modellje

A szigorú módban egy lebegőpontos típus előredefiniált műveleteinek eleget kell tenniük bizonyos kritériumoknak. Ezt a viselkedést a lebegőpontos aritmetika egy olyan modelljén keresztül definiálják, amely a korábban már bevezetett kanonikus formán alapszik.

Minden lebegőpontos típushoz tartozik a modellszámainak egy végtelen halmaza. Ezeket a modellszámokat használjuk a pontossági követelmények leírására. Egy származtatott típus modellszámai megegyeznek a szülő típus modellszámaival, míg egy altípus modellszámai megegyeznek a típusának modellszámaival.

Egy `T` lebegőpontos típus modellszámai a nulla, és minden olyan érték, amely kanonikus formában kifejezhető `T'Model_Mantissa` számjegyű mantisszával, és a kitevője nagyobb vagy egyenlő mint `T'Model_Emin`. (Ezeket az attribútumokat a későbbiekben fogjuk definiálni.)

Egy lebegőpontos típus egy modellintervalluma egy tetszőleges olyan intervallum, amelynek határai az adott típus modellszámai. Egy `T` típus `v` értékhez kapcsolódó modellintervalluma a legkisebb olyan modellintervallum, amely `v`-t tartalmazza.

Egy művelet egy paraméterének operandus-intervalluma az operandus típusának azon modellintervalluma, amely az operandus értékéhez kapcsolódik.

Bármely `T` lebegőpontos értékű előredefiniált aritmetikai művelet esetén az eredménnyel szemben megkövetelt korlátokat egy – az operandusoktól függő – modellintervallum (eredményintervallum) definiálja az alábbi módon: Az eredményintervallum a `T` típus legkisebb olyan modellintervalluma, amely tartalmazza azt a legkisebb és legnagyobb értéket, amelyet a pontos matematikai művelet végrehajtása eredményez az operandus-intervallum tetszőleges elemeire.

A hatványozás művelet eredményintervallumát úgy kaphatjuk, hogy a fenti szabályt alkalmazzuk az exponens által meghatározott szorzássorozatra, tetszőleges csoportosítást feltételezve. Negatív kitevő esetén a végső osztást is a műveletsorozat részének tekintjük.

Egy numerikus érték valamely `T` lebegőpontos típusra konvertálásának eredményintervalluma az operandus-intervallum lesz, kivéve, ha a kiindulási

kifejezés egy olyan fixpontos típusú érték, amelynek legkisebbje (Small) nem T'Machine_Radix hatványa, vagy ha a kifejezés az iménti tulajdonságú argumentumok szorzata vagy hányadosa. Ezekben az esetekben az eredményintervallum implementációfüggő.

A továbbiakban tárgyalandó műveletekre a nyelv egyes implementációinak akkor kell az eredményintervallumba eső értéket visszaadnia, ha az eredményintervallum határai beleesnek a T'Safe_First..T'Safe_Last intervallumba. Ellenkező esetben

- ha T'Machine_Overflows értéke True, akkor a nyelv megvalósítása vagy Constraint_Error kivételt vált ki, vagy visszaad egy, az eredményintervallumba eső értéket,
- ha T'Machine_Overflows értéke False, akkor az eredmény implementációfüggő.

Egy tetszőleges T lebegőpontos típus tetszőleges relációs operátora bármely olyan logikai értéket visszaadhat, amelyet a pontos matematikai összehasonlítás ad vissza az operandus-intervallumokból választott tetszőleges értékpárra.

A tartalmazásvizsgálat eredményét az összehasonlító operátorokon keresztül definiáljuk, a megadott intervallum, vagy típus alsó és felső korlátai és a megadott érték között.

G.2.2. Lebegőpontos típusok modellorientált attribútumai

Az AARM numerikus függelékét támogató implementációkban a lebegőpontos típusok modellorientált attribútumainak mind szigorú, mind laza módban az alábbi értékeket kell visszaadniuk (legyen S egy T lebegőpontos típus altípusa):

```
S'Model_Mantissa
```

Megadja a számjegyek számát a T típus modellszámai kanonikus alakjában. Az attribútum legalább Ceiling(d*log (10)/log (T'Machine_Radix))+1 kell legyen, ahol d a T típus megkívánt decimális pontossága. Ugyanekkor az attribútum értéke legfeljebb T'Machine_Mantissa. Az attribútum típusa univerzális egész.

```
S'Model_Emin
```

Megadja a T típus modellszámai kanonikus alakjában a legkisebb kitevő értékét. Az attribútum értéke legalább T'Machine_Emin kell legyen. Az attribútum típusa univerzális egész.

`S'Safe_First`

Megadja a `T` típus biztonságos intervallumának alsó határát. Az attribútum értéke a `T` típus egy modellszáma, és nagyobb vagy egyenlő mint a `T` típus alap értékkészletének alsó határa. Továbbá ha a `T` típus egy olyan valós intervallummal van definiálva, amelynek alsó határa `lb`, akkor az attribútum értéke legfeljebb `lb` lehet, míg ellenkező esetben legfeljebb -10^{4d}, ahol d a `T` típus megkívánt decimális pontossága. Az attribútum típusa univerzális valós.

`S'Safe_Last`

Megadja a `T` típus biztonságos intervallumának felső határát. Az attribútum értéke a `T` típus egy modellszáma, és kisebb vagy egyenlő mint a `T` típus alap értékkészletének felső határa. Továbbá ha a `T` típus egy olyan valós intervallummal van definiálva, amelynek felső határa `ub`, akkor az attribútum értéke legalább `ub` lehet, míg ellenkező esetben legalább 10^{4d}, ahol d a `T` típus megkívánt decimális pontossága. Az attribútum típusa univerzális valós.

`S'Model`

Az attribútum az alábbi specifikációval adott függvényt jelöli:

```
function S'Model (X : T) return T;
```

Ha `X` a `T` típus egy modellszáma, akkor a függvényérték `X`, ellenkező esetben pedig az `X`-szel szomszédos modellszámok valamelyike. Ha az eredményül kapott modellszám kívül esik az `S` biztonságos értékkészletén, akkor a függvény `Constraint_Error` kivételt vált ki. Ha `S'Signed_Zeros` értéke `True`, akkor nulla eredmény esetén az előjel megegyezik `X` előjelével.

A szigorú numerikus módban megkövetelt pontossági előírásokat – mivel azok elsősorban a nyelv implementálóinak szólnak –, itt nem részletezzük. Megjegyezzük azonban, hogy a pontos követelménydefiníció megtalálható az Annotated Ada Reference Manual G függelékében.

H. Függelék

A magasszintű programozási nyelvek alapfogalmai

A számítástudomány kérdéseinek elemzéséhez több modellt is kidolgoztak. [12]

H.1. A számítási modellek fajtái

Az *egyszerű operációs modell*ek közé soroljuk a Turing-gépet és az automatákat. Ennek a modellnek van megfelelő matematikai leírása, amelyre alapozni lehet. A modell definiál a program és az adatok számára egy memóriát, és a végrehajtást egyszerű állapotok sorozataként fogja fel. A programok viszont nem eléggé világos szerkezetűek.

A *Neumann-féle modell*-hez tartoznak a hagyományos programozási nyelvek és a Neumann-elvű számítógépmodell. Ennek a modellnek a leírása terjedelmes, szöveges. Ez a modell szintén definiál memóriát, a végrehajtást bonyolult állapotok sorozataként fogja fel, ahol az állapot a memória tartalmával fejezhető ki. Az adatok és a program nem válnak szét eléggé. A programok viszont világos szerkezetűek ebben a modellben.

A *funkcionális modell*-ek közé tartozik pl. a Church-féle λ-kalkulus és a LISP programozási nyelv. Ennek a modellnek van jó matematikai leírása. A funkcionális modell nem tartalmaz memóriát, a program statikus, egyik lépése sem ismétlődik. Az eredményt a program egyszerűbb programmá redukálásával kapjuk meg. A redukálásnak kétféle művelete van: alkalmazhatjuk a függvényt az argumentumaira, vagy kifejthetünk egy sorozatot (Trans,/,α) vagy függvénykompozíciót ($\circ$).

Az *objektum-orientált modell* – ahogy az elnevezés is mutatja – olyan szoftver-konstrukciós módszer, amely a programrendszert az általa használt adatok, a valós világ modellezésével kapott objektumok alapján építi fel.

E négy modellen kívül kialakultak még más modellek is, ilyenek pl. a különféle *logikai alapú* (pl. PROLOG) és egyéb *nemprocedurális* (pl. LUCID), vagy a *data flow modell*-ek.

H.2. Neumann-elvű nyelvek

A Neumann-elvű nyelveket Neumann-elvű gépekre tervezték. A Neumann-elvű nyelvek *utasítás-orientált nyelvek*: az utasítások egymás utáni vagy ismételt végrehajtásával kapjuk meg az eredményt. A Neumann-elvű nyelvek rendelkeznek a következő tulajdonságokkal:

a) A memória rekeszeinek absztrakciójaként a Neumann-elvű nyelvek bevezetik a *változó* fogalmát.

b) Szintén a memória következménye, hogy a kiszámított értékeket tárolni kell. Az „eltárolás" művelete az *értékadás*.

c) Az *utasítások ismételten végrehajthatók*. Ez annak a következménye, hogy a program is a memóriában helyezkedik el.

A Neumann-elvű nyelvek között nagyon sokféle nyelvet találunk.

H.3. A nyelvi elemek, alapfogalmak

A magasszintű programozási nyelvek története már több évtizedet ölel át. Ezalatt az idő alatt kialakult ezeknek a nyelveknek az általánosnak mondható szerkezete, megszülettek a leggyakoribb nyelvi alapegységek, kikristályosodtak az alapvető implementációs technikák és hagyományossá váltak egyes szemantikus szabályok. Ebben a pontban összefoglaljuk ezeket a fogalmakat.

H.3.1. Általános fogalmak

A nyelv *szintaxis*-ának nevezzük azoknak a szabályoknak az összességét, amelyek az adott nyelven írható összes lehetséges, formailag helyes programot (jelsorozatot) definiálják.

Az adott nyelv programjainak jelentését leíró (tehát azt, hogy a program végrehajtásának mi lesz a hatása) szabályok összességét a nyelv *szemantikájának* nevezzük.

A programok végrehajtására kétféle technika alakult ki: az interpreter és a fordítóprogram. Az *interpreter* egy utasítás lefordítása után azonnal végrehajtja azt, a *fordítóprogram* viszont átalakítja a programot egy vele ekvivalens formára, ami lehet a számítógép által közvetlenül végrehajtható

forma, vagy lehet egy másik programozási nyelv, s ekkor a program tovább fordítható vagy interpretálható. A lefordítandó programot *forrásprogram*-nak, a fordítás eredményeként kapott programot *tárgyprogram*-nak nevezzük. Azt az időt, amikor az utasítások fordítása folyik, *fordítási idő*-nek, azt pedig, amikor az utasítások végrehajtása folyik, *végrehajtási idő*-nek nevezzük.

H.3.2. Programszerkezet

A nyelvnek azt az egységét (azt a programrészt), amit a fordítóprogram egyszeri lefuttatásával, a program többi részétől elkülönülten lefordíthatunk, *fordítási egység*-nek nevezzük. Maga a program így fordítási egységek véges, nem üres halmaza.[1]. A fordítási egység szinonimájaként használatos a *modul* elnevezés is. (Több modulból álló programok esetében a fordítás és a végrehajtás közé egy újabb művelet lép, a program összeszerkesztése.)

A fordítási egység összetett nyelvtani egységek, ún. *programegység*-ek halmaza. Egyes nyelvekben a fordítási egység pontosan egy programegységből áll (pl. FORTRAN), másutt a fordítási egység programegységek sorozata (pl. C), ismét másutt lehetővé teszik a programegységek egymásba ágyazását is (pl. Modula-2). A programegység egy részfeladatot megoldó, tehát funkcionálisan összefüggő programrész. A programegységek alapvetően két részre bonthatók fel: a *specifikációs rész*-re és a *törzs*-re. A specifikációs rész írja le az egységnek a más egységből elérhető „kapcsolódási pontjait", a törzs tartalmazza az egység funkcióját megvalósító programot. A törzs szintén két fő részből áll: *deklarációs rész*-ből és *utasítássorozat*-ból. A deklarációs rész azonosítók deklarációit, valamint objektumok és programegységek definícióit tartalmazza. A törzs utasítássorozata tartalmazza azokat az utasításokat, amelyek *végrehajtása* nyomán a programegység kifejti a hatását.

Egyes nyelvekben a programegység szinonimájaként, másutt fajtájaként szerepel a *blokk* elnevezés is. A legelterjedtebb programegységek az *eljárások* és a *függvények*.

H.3.3. Deklarációk

A deklarációs rész azonosítók *deklarációi*t, valamint konstansok, típusok, objektumok és programegységek *definíciói*t tartalmazhatja. Egy deklaráció egymáshoz rendel egy azonosítót és egy definiált egyedet. (A leggyakoribb az, hogy az egyed definíciója egy deklarációs utasítás részeként szerepel.)

A deklarációs rész feldolgozását, a definíciók értelmezését *a deklarációs rész kiértékelésé*-nek (vagy a definíciók kiértékelésének) mondjuk.

[1]egyes programozási nyelvekben a program pontosan egy fordítási egységből áll

A deklarációs rész definícióinak kiértékelése történhet fordítási (pl. Modula-2) vagy futási (pl. PL/I, Ada) időben. Az első esetben *statikus definíció-kiértékelés*-ről, a másodikban *dinamikus definíció-kiértékelés*-ről beszélünk (ez utóbbi esetben lehetőség van pl. arra, hogy a típusok definíciói ne csak statikus kifejezésektől, hanem változóktól is függjenek).

A programszövegnek azt a szakaszát, amelyen belül az adott deklaráció érvényben van, a deklaráció *hatáskör*-ének nevezzük. A programszövegnek azt a szakaszát, ahol a deklarált azonosítóval az adott deklarációra hivatkozni lehet, a deklaráció *láthatósági kör*-ének nevezzük. (Világos, hogy a láthatósági kör része a hatáskörnek.)

A *blokkstruktúra* a programegységek egymásba ágyazásával előálló hierarchikus struktúra. Blokkstruktúrában egy programegység deklarációinak hatásköre kiterjed az összes tartalmazott programegységre. Egy programegységben a tartalmazó programegységek által deklarált azonosítókat (a tartalmazott programegység szemszögéből) *globális* azonosítóknak, a programegységben deklarált azonosítókat magában a programegységben *lokális* azonosítóknak nevezzük. Blokkstruktúra esetén egy programegységben valamely globális azonosító újradeklarálható, ekkor azt mondjuk, hogy ez a lokális deklaráció *eltakarja* a globális deklarációt. Az alábbi példán mutatjuk be, hogyan alakul a deklarációk hatásköre blokkstruktúra esetén:

```
   block M
     block P
       block A
         block B
         end B;
       end A;
     end P;
     block Q
       block R
       end R;
      block S
      end S;
    end Q;
  end M;
```

Az alábbi blokkokban deklarált azonosítók hatásköre a következő blokkokra terjed ki:

M	M, P, A, B, Q, R, S
P	P, A, B
A	A, B
B	B
Q	Q, R, S
R	R
S	S

A nyelvek általában lehetőséget adnak az azonosítók *átlapolásá*-ra (overloading), vagyis arra, hogy egy azonosítónak ugyanabban a szövegrészben több deklarációja is érvényben legyen. Az átlapolást a leggyakrabban alprogramok nevei esetében teszik lehetővé. (Jellegzetesen ilyen nevek pl. az aritmetikai műveletek előredefiniált nevei, a „+", a „/" stb.) Átlapolt azonosítók esetén a hivatkozás környezete teszi egyértelművé, hogy melyik deklarációra történt a hivatkozás.

Egy nyelvet *erősen típusosnak* nevezünk, ha minden érték, objektum, formális paraméter és függvény típusa fordítási időben egyértelműen meghatározható (ilyen nyelv pl. az Ada).

H.3.4. Memóriakezelés

A változók az operatív memóriában a típusuknak megfelelően különböző méretű helyet foglalnak el. A változóhoz a memóriaterület hozzárendelése (*allokálás*-a) a nyelvek egy részénél (többségénél) *automatikusan*, a definíció kiértékelésekor történik. A másik alkalmazott megoldás az, hogy a változó deklarálásakor pusztán a név lép életbe, de a változó maga „nem születik meg" (ilyenek pl. a PL/I *CONTROLLED* változói). Ilyenkor a programozó rendelkezhet arról, hogy az allokálás a program végrehajtásának mely pillanatában történjen meg. Az első esetben a változó által lefoglalt terület felszabadítása is automatikus, az utóbbiban ez is a programozó hatáskörébe tartozik.

A változókhoz a szükséges memóriaterület lefoglalása (allokálása) és annak felszabadítása közötti időt a változó *élettartam*-ának nevezzük. (Az élettartam egyes esetekben a hatáskörbe tartozó programrész végrehajtásának ideje, van azonban olyan eset is, pl. a pointerrel megcímzett változók esetén, amikor meghaladja ezt.) Ebből a szempontból háromféle változót különböztetünk meg, és ezeknek az elhelyezése három különböző memóriaterületen történik: a statikus, a dinamikus memóriaterületen és a végrehajtási veremben. Azokat a változókat, amelyek élettartama a program egész működése idejére kiterjed, *statikus változó*-knak nevezzük, és elhelyezésük mindig az ún. *statikus* (main) *memória*-területre történik. A statikus terület egyszer, a program betöltésekor kerül lefoglalásra. A *dinamikus* (heap) *memória*-területen a program explicit módon (ilyen célú szintaktikai elem segítségével) foglalhat le területet, amire a címével, ún. *pointer*-rel hivatkozhat. (Egyes nyelvek tartalmaznak utasítást a dinamikus területek felszabadítására is.) Blokkstruktúra esetén a programegységek (nem statikus) lokális változói ún. *végrehajtási verem*ben helyezkednek el. Egy speciális verem-teteje mutató jelzi a végrehajtási verem (aktuális) méretét. Egy programegység végrehajtásának megkezdésekor a verem-teteje mutató átállítódik, és a programegység lokális változói a mutató régi és új értéke közötti címeken helyeződnek el. A programegység végrehajtásának befejezésekor a verem-teteje mutató visszaáll a régi értékére. (Ez megadja annak

a lehetőségét, hogy egy alprogram – rekurzívan – önmagát végrehajthassa.) A memóriakezelésnek ez a három formája egyes nyelvekben egyszerre jelen van (pl. Ada, PL/I, Modula-2).

H.3.5. Paraméterátadás

A programegységek legelterjedtebb fajtája az *alprogram*. A nyelvek kétféle alprogramot használnak, ezek az *eljárás*ok és a *függvény*ek. Az alprogramok *formális paraméter*ekkel rendelkeznek, amelyeknek az *alprogram aktivizálásakor* (*hívás*akor) *aktuális paraméter*eket kell megfeleltetni. A *paraméterátadás*nak (vagyis az aktuális és formális paraméterek közötti megfeleltetésnek, értékcseréjének) négy formája alakult ki: a cím szerinti, a név szerinti, az érték szerinti és az eredmény szerinti paraméterátadás.

A *cím szerinti paraméterátadás* esetében (pl. FORTRAN) az aktuális paraméter vagy egy változó, vagy egy változó komponensét meghatározó kifejezés (pl. `x[i]`) lehet. Az aktuális paraméter kiértékelése a hozzá rendelt memóriaterület címének meghatározását jelenti. Az aktuális paraméter kiértékelésére az alprogram végrehajtásának megkezdése előtt kerül sor, s a formális paraméterhez az aktuális paraméter memóriaterülete rendelődik hozzá. Így a formális paraméter az alprogram végrehajtása során minden pillanatban azonos az aktuális paraméterrel. Az alprogramban hivatkozhatunk is a formális paraméter értékére, és adhatunk is neki új értéket.

Érték szerinti paraméterátadás esetén (pl. C) az aktuális paraméter egy kifejezés, a formális paraméter pedig az alprogram egy lokális változója, aminek definiált a kezdőértéke, és ez éppen az aktuális paraméter értéke. Az aktuális paraméter kiértékelésére egyszer, az alprogram végrehajtásának megkezdése előtt kerül sor. Az alprogram végrehajtása közben sem az aktuális paraméter értékének esetleges megváltozása (az aktuális paraméter értékének megváltoztatása pl. akkor történhetne meg, ha az aktuális paraméter egyben az alprogramra nézve globális változó is) nincs hatással a formális paraméter értékére, sem a formális paraméter értékének megváltoztatása nincs hatással az aktuális paraméterre.

Az *eredmény szerinti paraméterátadás* az alprogram befejezésének pillanatáig megegyezik az érték szerinti paraméterátadással. Az eredmény szerinti paraméterátadás annyival több, hogy az alprogram végrehajtásának befejezésekor az aktuális paraméter felveszi a formális paraméternek a pillanatnyi értékét.

Név szerinti paraméterátadás esetén (pl. ALGOL 60) az aktuális paraméter lehet változó (pl. `x`, `x[i]`), de lehet tetszőleges kifejezés is (pl. `x+y`). Az aktuális paraméter kiértékelése az első esetben a hozzá rendelt memóriaterület címének meghatározását, a másodikban a kifejezés értékének kiszámítását jelenti. A formális paraméter az alprogram végrehajtása során minden pillanatban azonos ezzel a változóval, illetve kifejezéssel. Az aktuális paraméter kiértékelése mindig újra megtörténik, valahányszor a

formális paraméterre hivatkozás történik. Tehát, ha az aktuális paraméter pl. `x[i]` vagy `k+j` alakú kifejezés, akkor az `i, k, j` változók értékének (a hívott alprogramból való) megváltoztatásával azonnal megváltozik a formális paraméter értéke is. (Ha az aktuális paraméter olyan kifejezés, ami nem szerepelhet értékadás baloldalán – pl. `k+j` –, akkor a formális paraméter nem kaphat új értéket az alprogram végrehajtása közben.)

I. Függelék

Feladatok

Az alábbi feladatok az Ada programozási nyelv használatának a gyakorlására szolgálnak. A feladatok többsége algoritmikus szempontból egyszerű, nehézséget csak az absztrakt programnak a nyelvre való átültetése okozhat.

Azokat a feladatokat, amelyek megoldása valamivel nehezebb, egy csillaggal (*), azokat pedig, amelyek hosszabb gondolkodást igényelnek, két csillaggal (**) jelöltük meg.

Felhívjuk a figyelmet arra, hogy a feladatok egy részéhez felsőfokú matematikai ismeretek szükségesek.

I.1. Készítsünk egy programot, ami kiszámítja a π közelítő értékét az alábbi képlet alapján!

a) $\frac{\pi}{2} = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \ldots$

b) $\pi = 4(1 - \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \ldots)$

c) $\pi = 3 + 4(\frac{1}{2 \cdot 3 \cdot 4} - \frac{1}{4 \cdot 5 \cdot 6} + \frac{1}{6 \cdot 7 \cdot 8} - \ldots)$

I.2. Készítsünk egy programot, ami ε pontossággal kiszámítja az alábbi közelítő értékeket!

a) $ln\ 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \ldots$

b) $ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots$

c) $ln\ x = 2\ (\frac{x-1}{x+1} + \frac{(x-1)^3}{3(x+1)^3} + \frac{(x-1)^5}{5(x+1)^5} + \ldots)$

I.3. Készítsünk egy programot, ami kinyomtatja a következő összeg első N részletösszegét!

a) $$\sum_{n=1}^{\inf} \frac{n!}{n^{\sqrt{n}}}$$

b) $$\sum_{n=1}^{\inf} \frac{((n+1)!)^n}{2!4!\dots(2n)!}$$

I.4. Függvények egy sorozatát az alábbi módon definiáljuk:

$$\begin{aligned} u_1(t) &= 1, \\ u_2(t) &= t, \\ u_{k+1}(t) &= 2tu_k(t) - u_{k-1}(t) \quad (k = 2, 3, ..., n-1) \end{aligned}$$

Készítsünk programot, ami adott n és t értékekre meghatározza $u_n(t)$ függvényértéket!

I.5. Készítsünk olyan eljárást, ami az adott n, h egész és y_0 valós értékekből meghatározza y_n értékét az alábbi formula alapján:

$$\begin{aligned} &y_{i+1} = y_i + \frac{1}{4}k_i + \frac{3}{4}m_i \quad (0 \le i < n), \\ &k_i = h(y_i^2 + y_i + 1), \\ &m_i = h((y_i + \frac{2}{3}k_i)^2 + y_i + \frac{2}{3}k_i + 1) \end{aligned}$$

I.6. Adott a valós A_N vektor. Határozzuk meg S értékét az alábbi formula alapján:

a) $$S = \frac{1}{N-1} \sum_{i=1}^{N} (a_i - \frac{1}{N} \sum_{j=1}^{N} a_j^2)$$

b) $$S = \sqrt{\frac{1}{N} \sum_{i=1}^{N} a_i^2 - (\frac{1}{N} \sum_{j=1}^{N} a_j)}^{\,2}$$

I.7. Készítsünk programot az alábbi összeg kiszámítására (a_0, b_0 és ε adott kezdőértékek):

$$S = 1 - \frac{1}{2} \sum_{n=0}^{a_n^2 - b_n^2 < \varepsilon} 2^n (a_n^2 - b_n^2), \quad \textit{ahol}$$

$$a_n = \frac{1}{2}(a_{n-1} + b_{n-1}), \quad b_n = \sqrt{a_{n-1} b_{n-1}}$$

I.8. Adottak a valós X_n és Y_n vektorok. Készítsünk programot az a_k együtthatók kiszámítására az alábbi formula alapján:

$$a_k = \frac{4}{n\pi} \sum_{i=1}^{n} \frac{y_i u_k(x_i)}{\sqrt{1-x_i^2}}, \qquad k = 1, ..., n,$$

$$u_1(z) = 1, \quad u_2(z) = z, \quad u_{k+1}(z) = 2zu_k(z) - u_{k-1}(z), \quad k = 2, ..., n$$

I.9. Készítsünk programot az alábbi összeg kiszámítására:

$$a) \quad S = 1 + x + \frac{x^2}{2!} - \frac{3x^4}{4!} + ... + (-1)^{N-1} \frac{(2N-1)x^{2N}}{(2N)!},$$

$$b) \quad S = \frac{2}{3} sin\ 2x - \frac{3}{8} sin\ 3x + ... + (-1)^N \frac{N}{N^2-1} sin\ Nx$$

I.10. Készítsünk olyan programot, ami kiszámítja az alábbi tört értékét!

$$\frac{1}{1 + \frac{1}{3 + \frac{1}{...n + \frac{1}{n+1}}}}$$

I.11. Készítsünk olyan programot, ami ε pontossággal kiszámítja $\sqrt[k]{x}$ értékét az alábbi rekurzív formula alapján :

$$a) \quad y_0 = x, \quad y_{n+1} = \frac{1}{k}((k-1)y_n + \frac{x}{y_n^{k-1}})$$

$$b) \quad y_0 = x, \quad y_{n+1} = \frac{y_n}{k^2}((k^2-1) + \frac{1}{2}(k+1)\frac{x}{y_n^k} - \frac{1}{2}(k-1)\frac{y_n^k}{k})$$

I.12. Adottak egy valós vektorban egy polinom együtthatói.

a) Készítsünk olyan programot, ami kiszámítja a deriváltjának együtthatóit!

b) Készítsünk olyan programot, ami kiszámítja az integráljának együtthatóit!

c) Készítsünk olyan programot, ami kiszámítja a négyzetének együtthatóit!

I.13. Készítsünk olyan eljárást, amelynek bemenő paraméterei egy m-ed és egy n-ed rendű polinom együtthatóit tartalmazó vektorok, kimenő paramétere egy $m+n$ hosszúságú vektor, s az eljárás kiszámítja a szorzatpolinom együtthatóit!

I.14. Készítsünk olyan csomagot, ami polinomokra vonatkozó eljárásokat exportál (polinomok összeadása, kivonása, számmal való szorzása stb.)!

I.15. Adott a valós A szám. Készítsünk olyan programot, ami meghatározza az S és N számokat úgy, hogy $S*N$ minimális legyen, és $\mid \frac{S}{N} - A \mid < \varepsilon$ teljesüljön!

I.16. Készítsünk olyan programot, ami ε pontossággal kiszámítja $tg\ z$ értékét az alábbi rekurzív formula alapján (feltesszük, hogy $z \neq \frac{\pi}{2} \pm n\pi (n = 1, 2, ...))$:

$$tg\ z = \frac{z}{1 - y_3}, \qquad y_i = \frac{z^2}{i - y_{i+2}}$$

I.17. Készítsünk olyan programot, ami kiszámítja C_n^p értékét az alábbi rekurzív formulák alapján:

$$C_1^1 = 1, \quad C_n^{p+1} = \frac{n-p}{p+1} C_n^p, \quad C_{n+1}^p = C_n^{p-1} + C_n^p$$

I.18. Készítsünk csomag sablont a komplex számok kezelésére! Az elemi aritmetikai operációkon kívül készítsünk függvényeket az alábbiak kiszámítására is (a egy komplex számot jelöl, (r, ϕ) az a polárkoordinátás alakja, n egy egész szám):

$$\sqrt[n]{a} = \sqrt[n]{r} \cdot (cos \frac{\phi + 2k\pi}{n} + i \cdot sin \frac{\phi + 2k\pi}{n}), \quad (k = 0, ..., n - 1)$$

$$a^n = e^{n \cdot ln\ a}$$

$$sin\ a = a - \frac{a^3}{3!} + \frac{a^5}{5!} - \frac{a^7}{7!} + \ldots$$

$$cos\ a = 1 - \frac{a^2}{2!} + \frac{a^4}{4!} - \frac{a^6}{6!} + \ldots$$

I.19. Készítsünk egy olyan csomagot, ami egész számpárokkal ábrázolt valós számok műveleteit tartalmazza! Az (a, b) szám értéke $a \cdot 10^b$, tehát pl. az (a, b) és a (c, d) számok szorzata $(a \cdot c, b + d)$.

I.20. Készítsünk olyan programot, ami meghatározza azoknak az egész koordinátájú pontoknak a számát, amelyek az $\frac{y^2}{16} + \frac{x^2}{25} + 1 = 0$ ellipszisbe esnek!

I.21. Készítsünk olyan programot, ami szögfüggvénytáblázatot nyomtat!

I.22. Készítsünk egy olyan programot, ami kinyomtatja az alábbi függvény értékeinek táblázatát az $[a, b]$ intervallumban 0.5 lépésközzel!

$$a)\quad f(x) = \frac{x}{(1+x)^2}$$

$$b)\quad f(x) = \frac{1+x}{1+x^3}$$

$$c)\quad f(x,n,p) = x^n p^x (1-p)^{n-x}$$

I.23. Ha egy testet elengedünk, szabadon esni kezd. Az elengedéstől számított t. másodpercben a kiindulási ponttól mért távolságát lábban az $s = 16 * t^2$ képlet adja (1 láb=30.48 cm). Nyomtassunk táblázatot az 1–20 másodpercekről, és ebben adjuk meg a test által megtett utat cm-ben!

I.24. Készítsünk egy olyan programot, ami a tanult közelítő eljárások valamelyikének felhasználásával megkeresi az alábbi egyenlet egy gyökét (pl. intervallumfelezés, húrmódszer)!

$$a)\quad x^3 - 2x + 1.2 = 0, \qquad a\ [-2,-1]\ intervallumban$$

$$b)\quad x^5 - 0.5 = 0, \qquad a\ [0,1]\ intervallumban$$

$$c)\quad e^x - x^2 + 1 = 0, \qquad a\ [-2,-1]\ intervallumban$$

$$d)\quad e^x - x^2 - 2 = 0, \qquad a\ [0.5, 0.6]\ intervallumban$$

I.25. Készítsünk egy olyan programot, ami a tanult közelítő eljárások valamelyikének felhasználásával kiszámolja az alábbi függvény határozott integrálját (pl. Simpson-módszer, trapéz formula)!

$$a)\quad \int_{-1}^{1} (1 + sin\ x)^2$$

$$b)\quad \int_{-1}^{1} x^2 + \sqrt{x^2 + 0.1}$$

$$c)\quad \int_{-1}^{1} \frac{x}{1+x^2}$$

$$d)\quad \int_{-2}^{-1.5} x^3 - 3x^2 - x + 9$$

I.26. Készítsünk programot, ami Monte-Carlo módszerrel kiszámítja az $(x-0.5)^2 + (y-0.5)^2 < 0.25$ körlap területét! (Állítsunk elő kb. 200 pontot egyenletes eloszlással véletlenszerűen a $[-1,1]\times[-1,1]$, tartományban, majd vizsgáljuk meg, hogy a pontok közül hány esik a körlapra.)

I.27. Készítsünk programot, ami Monte-Carlo módszerrel kiszámítja egy függvény határozott integrálját! (Foglaljuk téglalapba a függvényt az intervallum határai és a függvény egy alsó- és egy felső korlátja segítségével. Állítsunk elő egyenletes eloszlással véletlenszerűen pontokat, s állapítsuk meg, hogy a függvénygörbe alatt vagy felett vannak-e. Ezután egy osztással megkapható a görbe alatti terület közelítése.)

I.28. Készítsünk egy olyan programot, ami a tanult eljárások valamelyikének felhasználásával megoldja az alábbi egyenletet!

$$a)\quad y' = x + cos\frac{y}{\pi},\ x_0 = 1.7,\ y_0 = 5.3,\ az\ [1.7, 5.2]\ intervallumban$$

$$b)\quad y' = x - y,\ x_0 = 0,\ y_0 = 0,\ a\ [0, 1]\ intervallumban$$

$$c)\quad y' = \frac{y}{e^x + y^2},\ x_0 = 0,\ y_0 = 1,\ a\ [0, 1]\ intervallumban$$

I.29. Készítsünk egy olyan programot, ami a Runge-Kutta módszer felhasználásával megoldja az alábbi egyenletet!

$$a)\quad y' = 0.25y^2 + x^2,\ y(0) = -1,\ a\ [0, 0.5]\ intervallumban$$

$$b)\quad y' = \frac{y}{x} - y^2,\ y(0) = 1,\ az\ [1, 2]\ intervallumban$$

I.30. Készítsünk egy olyan programot, ami az Euler-módszer felhasználásával megoldja az alábbi egyenletet!

$$a)\quad y' = \frac{1}{2}xy,\ y(0) = 1,\ a\ [0, 1]\ intervallumban$$

$$b)\quad y' = 1 + xy^2,\ y(0) = 0,\ a\ [0, 1]\ intervallumban$$

$$c)\quad y' = ax + by + c,\ ahol\ a, b, c, y(0)\ valamint\ az\ intervallum\ adottak$$

I.31. Készítsünk egy olyan programot, ami a Newton-módszer ($x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$) felhasználásával megoldja az alábbi egyenletet!

$$a)\quad log_{10}x - \frac{1}{x^2} - 1 = 0,\quad ha\ x_0 = 4$$

$$b)\quad cos\ x - 3x = 0,\quad ha\ x_0 = 0.3$$

$$c)\quad e^x - 2x - 21 = 0,\quad ha\ x_0 = 3$$

I.32. Készítsünk egy olyan programot, ami a Seidel-módszer felhasználásával megoldja az alábbi egyenletrendszert!

$$a)\quad \begin{array}{rcrcrcl} 4x_1 & + & 0.24x_2 & - & 0.08x_3 & = & 8, \\ 0.09x_1 & + & 3x_2 & - & 0.15x_3 & = & 9, \\ 0.04x_1 & - & 0.08x_2 & + & 4x_3 & = & 20. \end{array}$$

$$b)\quad \begin{array}{rcrcrcl} 10x_1 & + & x_2 & + & x_3 & = & 12, \\ 2x_1 & + & 10x_2 & + & x_3 & = & 13, \\ 2x_1 & + & 2x_2 & + & 10x_3 & = & 14. \end{array}$$

I.33. Készítsünk egy olyan programot, ami a Gauss-módszer felhasználásával megoldja az alábbi egyenletrendszert!

$$a)\quad \begin{array}{rcrcrcrcl} 7.9x_1 & + & 5.6x_2 & + & 5.7x_3 & - & 7.2x_4 & = & 6.68, \\ 8.5x_1 & - & 4.8x_2 & + & 0.8x_3 & + & 3.5x_4 & = & 9.95, \\ 4.3x_1 & + & 4.2x_2 & - & 3.2x_3 & + & 9.3x_4 & = & 8.6, \\ 3.2x_1 & - & 1.4x_2 & - & 8.9x_3 & + & 3.3x_4 & = & 1. \end{array}$$

$$b)\quad \begin{array}{rcrcrcl} 6x_1 & - & x_2 & - & x_3 & = & 11.33, \\ -x_1 & + & 6x_2 & - & x_3 & = & 32, \\ -x_1 & - & x_2 & + & 6x_3 & = & 42. \end{array}$$

I.34. Készítsünk olyan programot, ami eldönti, hogy az

$$\begin{array}{rcl} x_1 & = & (5,-3,2,4), \\ x_2 & = & (2,-1,3,5), \\ x_3 & = & 4,-3,5,-7) \end{array}$$

vektorok lineárisan függetlenek-e! (A feladat visszavezethető a $\lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3 = 0$ egyenlet megoldására. Használjuk erre a Gauss-Jordan módszert!)

I.35. Készítsünk olyan programot, ami kiírja adott A_0 tőke 1,2,...,10 évi kamatos kamattal növelt értékeinek táblázatát (p a kamatláb) :

$$A = A_0(1 + \frac{p}{100})^n$$

I.36. Készítsünk egy olyan programot, ami kinyomtatja egy adott év adott hónapjának naptárát!

I.37. Az egyetemen a pénztár olyan címletekben fizeti ki a hallgatók ösztöndíjait a csoportvezetőnek, hogy az mindenkinek odaadhassa az ösztöndíjat további pénzváltás és -visszaadás nélkül, s közben a pénztár a lehető legkevesebb bankjegyet és érmét adja ki. Készítsük el a címletezés programját, aminek adata a csoport tagjainak ösztöndíjai, eredménye az, hogy hány darab 1000-, 500-, 100-, 50-, 20-, 10-, 5-, 2- és 1-forintost kell a csoportvezetőnek kiadni!

I.38. Adott egy 1582 és 4902 közötti dátum (év, hónap, nap). Készítsünk programot, ami megállapítja, hogy ez hányadik nap az évben!

I.39. Készítsünk programot, ami megállapítja, hogy egy adott dátum hányadik munkanap az évben, ill. megfordítva, ami meg tudja adni, hogy az n. munkanap az év melyik napjára esik! (Az ünnepnapok egy része — pl. a karácsony — ismert, mások egy táblázatban adottak.)

I.40. Készítsünk egy csomagot a dátum (év, hónap, nap) típus megvalósítására, a dátumok közötti aritmetikai műveletek (dátum + valahány nap, két dátum között eltelt napok száma, két dátum közül melyik a korábbi stb.) elvégzésére!

I.41. Készítsünk programot, ami meghatározza egy adott évben az összes olyan pénteki napot, ami 13-ára esik!

I.42. Készítsünk programot, ami a természetes számok számjegysorozatában meghatározza az n. számjegyet!

I.43. Készítsünk programot, ami megszámolja, hogy az m és n között elhelyezkedő természetes számok összesen hány számjegyből állnak!

I.44. Készítsünk olyan programot, ami meghatározza azokat a háromjegyű természetes számokat, amelyek számjegyeinek köbösszege kiadja magát a számot!

I.45. Készítsünk olyan programot, ami meghatározza azokat a természetes számokat, amelyekre teljesül, hogy felírásukban és négyzetük felírásában együttesen a 0 -t kivéve minden számjegy szerepel, de csak egyszer!

I.46. Készítsünk olyan programot, ami meghatározza azokat a d_i természetes számokat, amelyekre teljesül, hogy $d_n \neq 0$ és $\forall i \in 0, ..., n 0 \leq d_i \leq (i+1)!$, valamint $\sum_{i=0}^{n} d_i \cdot (i+1)!$ egy előre megadott számmal egyenlő!

I.47. Készítsünk programot, ami egy adott N természetes számot felbont két prímszám összegére (nem bizonyított, hogy ez mindig lehetséges)!

I.48. Bármely természetes szám előállítható legfeljebb négy természetes szám négyzetösszegeként. Készítsünk programot, ami egy adott N természetes számhoz megadja azokat a természetes számokat, amelyek négyzetösszegeként előállítható!

I.49. Készítsünk programot, ami egy adott N természetes számról megállapítja, hogy előállítható-e három természetes szám négyzetösszegeként, s ha igen, akkor felsorolja az összes ilyen előállítást!

I.50. Készítsünk programot, ami egy adott N természetes számhoz megadja az N -nél kisebb $2^p - 1$ alakú természetes számokat, ahol p prímszám!

I.51. Készítsünk programot, ami meghatározza azt a legnagyobb N természetes számot, amire $N! \leq 10^{50}$!

I.52. Tökéletesnek nevezzük azokat a természetes számokat, amelyek osztóinak (az 1 -et beleértve, de önmagát nem) összege kiadja magát a számot. Készítsünk programot, ami M és N között megadja az összes tökéletes számot!

I.53. Ismert, hogy az m és n nemnegatív egész számok ($m \geq n$) legnagyobb közös osztója meghatározható az alábbi rekurzív függvény értékének kiszámításával:

$$lnko(m,n) = \begin{cases} m & , ha\ n = 0 \\ lnko(n, [\frac{m}{n}]) & , ha\ n \neq 0 \end{cases}$$

a) Készítsünk programot két természetes szám legnagyobb közös osztójának meghatározására!
b) Készítsünk programot n db természetes szám legnagyobb közös osztójának meghatározására! ($lnko(a_1, ..., a_n) = lnko(lnko(a_1, ..., a_{n-1}), a_n)$.)
c) Készítsünk programot két természetes szám legkisebb közös többszörösének meghatározására!

I.54. Adott az N természetes szám. Készítsünk programot, ami megállapítja, hogy
a) igaz-e, hogy a számban minden számjegy különböző!
b) igaz-e, hogy a számban egy számjegy háromszor szerepel, a többi számjegy pedig mind különböző!

I.55. Készítsünk programot egy személyi szám ellenőrző jegye helyességének megállapítására! A személyi számok utolsó jegye az ellenőrző jegy, amit az alábbi képlettel határoztak meg (10-es maradékot adó személyi számot nem adtak ki):

$$(x_1 + 2 \cdot x_2 + 3 \cdot x_3 + \ldots + 10 \cdot x_{10})\ mod\ 11$$

I.56. Készítsünk programot, ami prímtényezőire bontja az N természetes számot (vagyis meghatározza azokat az I, J, K, L, P, Q, R, S természetes számokat, amelyekre $N = I^P J^Q K^R ... L^S$)!

I.57. Adott egy 1-nél nagyobb természetes szám. Ha páros, akkor felezzük, különben szorozzuk meg hárommal, majd adjunk hozzá egyet, s ismételjük ezt a műveletet, amíg 1 -et nem kapunk! Készítsünk programot, ami a fenti eljárást alkalmazza, s megadja, hogy hány lépés után jutott el az 1-hez! (Nem bizonyított, hogy ez az eljárás minden természetes számra véges.)

I.58. Készítsünk programot, ami meghatározza az $a_0, a_1, ..., a_n$ számsorozat elemeit ($a_i \in \{-1, 0, 1\}, a_n \neq 0$) úgy, hogy $\sum_{i=0}^{n} a_i 3^i$ egy előre megadott érték legyen!

I.59. Készítsünk programot, ami előállít egy olyan természetes számot, aminek decimális alakjában csak a 0 és a 7 számjegyek fordulnak elő, s a szám osztható egy előre megadott természetes számmal!

I.60. Készítsünk csomagot tetszőleges elemekből álló sorozat típus megvalósítására!

I.61. Készítsünk programot lineáris egyenletrendszer megoldására a trianguláris felbontás módszerével!

I.62. Készítsünk olyan csomagot, ami egy tetszőleges elemekből álló vektor rendezésére exportál különböző eljárásokat (quicksort, heapsort stb.)!

I.63. Készítsünk olyan csomagot, ami numerikus elemekből álló vektorokon értelmezett műveleteket exportál (összeadás, szorzás, norma számolása, merőlegesség ellenőrzése stb.)!

I.64. Készítsünk olyan csomagot, ami egy általános mátrix típusra exportálja a gyakoribb mátrixműveleteket (mátrixok összeadása, kivonása, szorzása, számmal való szorzása, mátrixokon értelmezett relációk stb.)!

I.65. Készítsük el a főátlóra szimmetrikus mátrixok típusának egy hely szempontjából gazdaságos reprezentációját, s írjuk meg a típusműveletek alprogramjait (indexelés, mátrixok összeadása, kivonása, szorzása, számmal való szorzása, mátrixokon értelmezett relációk stb.)!

I.66. Készítsük el egy ritka (hézagosan kitöltött) mátrix típusnak a hely szempontjából gazdaságos reprezentációját, s írjuk meg a típusműveletek alprogramjait (indexelés, mátrixok összeadása, kivonása, szorzása, számmal való szorzása, mátrixokon értelmezett relációk stb.)!

I.67. Készítsünk olyan programot, ami meghatározza egy vektor érték szerint középső elemét!

I.68. Adott egy egész számokból álló vektor és egy K egész szám. Készítsünk olyan programot, ami átrendezi a vektor elemeit úgy, hogy a K-nál kisebb elemek állnak elöl!

I.69. Készítsünk egy programot, ami egy mátrixban felcserél két sort és két oszlopot úgy, hogy a mátrix maximális eleme az első sor első eleme legyen!

I.70. Készítsünk olyan eljárást, ami a paraméterként kapott négyzetes mátrixot az óra járása szerint 90 fokkal elforgatja!

I.71. Készítsünk olyan programot, ami a $2n$ méretű A mátrixot blokkokra osztja $A = \begin{matrix} A_1 & A_2 \\ A_3 & A_4 \end{matrix}$ alakban, majd a blokkokat felcseréli: A_1-et A_4-el, és A_2-t pedig A_3-mal!

I.72. Az A_N vektor természetes számokat tartalmaz. Határozzuk meg azoknak az elemeknek az indexét, amelyek összege megegyezik egy előre adott értékkel!

I.73. Az A_N vektor természetes számokat tartalmaz. Határozzuk meg azoknak az elemeknek az indexét, amelyek (a vektorbeli sorrendjükben) a leghosszabb növekvő sorozatot alkotják!

I.74. Készítsünk olyan eljárást, ami az adott X_n vektor ismeretében meghatározza a P_n vektor értékét úgy, hogy P az $1,\ldots,n$ számok permutációja legyen és teljesüljön $x_{p_i} \leq x_{p_{i+1}}$ $(i = 1, \ldots, n-1)$!

I.75. $**$ Adott egy valós mátrix és egy R valós szám. Bontsuk a mátrixot a lehető legkevesebb részmátrixra úgy, hogy egy részmátrixon belül bármely két elem különbsége kisebb legyen R -nél!

I.76. Készítsünk programot, ami kiszámolja egy mátrix sajátértékeit és sajátvektorait a Jacobi-módszer alapján!

I.77. Készítsünk csomag sablonokat különféle numerikus módszerekre!

I.78. Készítsünk olyan programot, ami egy négyzetes mátrixot felbont két mátrix összegére úgy, hogy az egyik tag szimmetrikus, a másik ferdén szimmetrikus legyen!

$$A = S + T, \quad S = \frac{1}{2}(A + A^*), \quad T = \frac{1}{2}(A - A^*)$$

I.79. Készítsünk olyan programot, ami egy négyzetes mátrixról eldönti, hogy az elemek átrendezésével szimmetrikus mátrixszá tehető-e!

I.80. Legyen $A^{(1)}, A^{(2)}, \ldots$ négyzetes bináris mátrixok egy sorozata, ahol $A^{(i)}$ dimenziója 3^i. Készítsünk programot az $A^{(i)}$ mátrix előállítására, ha

$$A^{(1)} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}, \qquad A^{(i)} = \begin{pmatrix} A^{(i-1)} & 0 & A^{(i-1)} \\ 0 & A^{(i-1)} & 0 \\ A^{(i-1)} & 0 & A^{(i-1)} \end{pmatrix}!$$

I.81. $*$ Adott az egészekből álló C_{mn} mátrix és a D_m vektor. Készítsünk programot annak a csak 0 és 1 számokat tartalmazó X_n vektornak az előállítására, amelyre $C \cdot X = D$!

I.82. Adott az S_N számsorozat. Egészítsük ki a sorozatot további M darab szám hozzávételével $(s_{N+1}, ..., s_M)$ úgy, hogy teljesüljön az alábbi feltétel: $s_1 = s_M, s_2 = s_{M-1}, ...$!

I.83. Adott a síkon koordinátáival N darab pont. Kössük össze a pontokat egy olyan vonallal, ami mindegyik ponton (pontosan egyszer) átmegy, s nem metszi önmagát!

I.84. Az $y = ax+b_1$ és az $y = ax+b_2$ egyenesek a síkot három tartományra osztják. Készítsünk olyan programot, ami megállapítja, hogy adott N darab pont közül melyik hányadik tartományba esik!

I.85. Adott a síkon N db kör. Készítsünk olyan programot, ami megállapítja, hogy a körök közös része üres-e!

I.86. Adott végpontjai koordinátáival a síkon két szakasz. Határozzuk meg a metszéspont koordinátáit!

I.87. Adott koordinátáival a sík N pontja. Rakjuk sorba a pontokat a rajtuk és az origón áthaladó egyenesnek az x - tengellyel bezárt szöge szerint növekvő sorrendbe!

I.88. Adott koordinátáival a sík N pontja. Határozzuk meg a ponthalmaz konvex burkát!

I.89. Adott koordinátáival a sík N pontja. Határozzuk meg a ponthalmazhoz a legkisebb olyan négyzetet, amelybe mindegyik pont beleesik!

I.90. Adott koordinátáival a sík N pontja. Készítsünk programot, ami kiválaszt három olyan pontot, amelyek (legfeljebb ε eltérést megengedve) egy egyenesre esnek!

I.91. Adott a síkon csúcspontjai koordinátáival egy sokszög (a csúcspontokat a kerületen az óra járása szerint elfoglalt helyük sorrendjében adjuk meg). Készítsünk programot, ami eldönti, hogy a sokszög konvex-e!

I.92. Készítsünk egy olyan programot, ami egy alfanumerikus képernyőre valamilyen jelekből (pl. csillagokból) egy adott sugarú kört rajzol!

I.93. Készítsünk egy olyan programot, ami egy alfanumerikus képernyőre valamilyen jelekből (pl. csillagokból) egy olyan egyenest rajzol, ami áthalad az origón, s aminek pontjai egyenlő távolságra vannak az A és a B ponttól!

I.94. Készítsünk egy olyan programot, ami egy alfanumerikus képernyőre valamilyen jelekből (pl. csillagokból) kirajzolja egy függvény grafikonját!

a) $y = \dfrac{x^4+8}{x^3+1}$, *a* $[-1,1]$ *intervallumban*

b) $y = sin\ x^2$, *a* $[0,\pi]$ *intervallumban*

I.95. Egy véletlenszámgenerátort ellenőrzünk abból a szempontból, hogy a (0,1) intervallumban eléggé egyenletesen generálja-e a számokat. Ehhez az intervallumot k egyenlő részre osztjuk, majd generálunk n db véletlenszámot. Rajzoljunk hisztogramot!

I.96. Ha r_i-k egyenletes eloszlású véletlenszámok a $[-1,1]$ intervallumban, akkor az alábbi képletekkel megadott pontok egyenletes eloszlásúak az egységgömbön. Készítsünk programot, ami ellenőrzi, hogy a pontok eloszlása valóban egyenletes-e!

$$x = \frac{2(r_2r_4 + r_1r_3)}{r_1^2 + r_2^2 + r_3^2 + r_4^2},$$

$$y = \frac{2(r_3r_4 - r_1r_2)}{r_1^2 + r_2^2 + r_3^2 + r_4^2},$$

$$z = \frac{r_1^2 + r_4^2 - r_2^2 - r_3^2}{r_1^2 + r_2^2 + r_3^2 + r_4^2},$$

$$\textit{ahol} \sum_{i=1}^{4} r_i^2 < 1.$$

I.97. Adott a síkon szemközti csúcsai koordinátáival egy téglalap, amelynek oldalai párhuzamosak a koordináta-tengelyekkel, és adott a végpontjai koordinátáival egy szakasz. Készítsünk olyan programot, ami kiszámítja a szakasznak azt a darabját, ami a téglalapon belülre esik (lehet, hogy a teljes szakasz kívül esik a téglalapon)!

I.98. * Rendelkezésünkre áll a Balaton vízfelszíne igen sok pontjának azonos időben mért hőmérséklete. Készítsünk programot a hőmérsékletek szintvonalas ábrázolására!

I.99. Készítsünk olyan csomagot, ami szövegben egy adott szöveg megkeresésére különböző eljárásokat exportál!

I.100. Készítsünk olyan csomagot, ami új típusműveleteket ad a `STRING` típushoz (karakter vagy sztring keresése sztringben, az összes szóköz kihagyása a sztringből, kivéve azokat, amelyek nem szóköz után következnek, sztringek összehasonlítása a szóközök figyelmen kívül hagyásával, sztringbe egy sztring beszúrása, sztringből egy részsztring elhagyása stb.)!

I.101. Készítsünk rekurzív eljárást, ami
a) egy szövegből kihagyja a szóközöket!
b) egy szövegben megszámolja a magánhangzókat!

I.102. Készítsünk programot, ami egy 4000-nél kisebb természetes számot római számmá alakít át!

I.103. Készítsünk olyan csomagot, ami római számokon végez el aritmetikai műveleteket!

I.104. Készítsünk programot, ami egy szövegben megkeresi a magas-, mély- és vegyes hangrendű szavakat!

I.105. $*$ Készítsünk programot, ami egy verset úgy nyomtat ki, hogy bejelöli az időmértéket!

I.106. $**$ Készítsünk egy interaktív programot, ami a beolvasott szavakat szótagolva írja vissza!

I.107. $*$ Készítsünk egy olyan programot, ami egy természetes számot betűkkel kiír! (Tehát pl. 241 esetén a Kettőszáznegyvenegy szöveget nyomtatja.)

I.108. Az egyik legegyszerűbb titkosírást az ABC betűinek ciklikus eltolásából kaphatjuk meg. Készítsünk olyan programot, ami egy szöveget kódol úgy, hogy minden betű helyébe az ABC-ben nála adott hellyel hátrább levő betűt teszi!

I.109. Egy titkosírást a következő módon készítünk el: helyezzünk el betűket egy négyzetes mátrixban, s erre tegyünk rá egy négyzetrácsos lapot, amelynek négyzetei egy-egy betű fölé esnek, s amelynek néhány négyzete ki van vágva. A látható betűk adják a megfejtés első részét. Ezután fordítsuk el ezt a lapot 90 fokkal jobbra, és folytassuk a betűk leolvasását. Ezt a műveletet még kétszer megismételve, megkapjuk a teljes szöveget. Készítsünk programot, ami a fenti módon megfejt egy titkosított szöveget!

I.110. A titkosírás egy módja a következő is: írjuk be a szöveget mondjuk sorfolytonosan egy mátrixba, majd a középső elemtől elindulva és spirálisan kifelé tartva járjuk be a mátrix elemeit. Az így kapott karaktersorozat a titkosított szöveg. Készítsünk egy programot, ami a fenti módon titkosít és megfejt egy szöveget!

I.111. Készítsünk egy programot, ami egy véletlenszámgenerátor felhasználásával titkosít egy szöveget! Inicializáljuk a véletlenszámgenerátort egy olyan értékkel, amit megjegyzünk, s amire szükségünk lesz majd a megfejtéskor is. Állítsunk elő pontosan annyi egyenletes eloszlású véletlenszámot 1 és 4 között, mint amilyen hosszú a szöveg. Ezután a szöveg minden karakterét cseréljük fel azzal a karakterrel, amit úgy kapunk, hogy a neki megfelelő véletlenszámot hozzáadjuk. Megfejtéskor ugyanazt a véletlenszámgenerátort kell használnunk, s ugyanazzal az értékkel kell inicializálni.

I.112. $*$ Egy text-file-ban a szöveget üres sorokkal tagoltuk bekezdésekre. Készítsünk olyan programot, ami úgy nyomtatja a file tartalmát, hogy az egyes bekezdések első sora előtt három szóközt hagy, s a bekezdéseken belül a sorokat egyformán 60 hosszúságúra hozza (ez történhet szavak áthelyezésével a következő sorba, a következő sorból szavak áthozásával, szóközök beszúrásával, esetleg szavak elválasztásával)!

I.113. Készítsünk programot, ami egy karaktersorozatról eldönti, hogy az a Modula-2 szintaxisa szerint egy szabályos felsorolási típus definíció-e!

I.114. Adott egy input rekordban egy zárójelezett kifejezés (maga az egész kifejezés is zárójelben van).
a) Készítsünk olyan programot, ami eldönti, hogy a zárójelezés helyes-e!
b) Készítsünk olyan programot, ami megadja (kiírja) az összes zárójelezett részkifejezést!

I.115. Készítsünk olyan programot, ami előállítja az összes helyes N hosszúságú zárójelezést! Helyes zárójelezés a (), továbbá, ha P és Q helyes zárójelezések, akkor helyes zárójelezés (P) és $(P)(Q)$ is.

I.116. Adott egy kifejezés, ami egybetűs vátozókból, zárójelekből és bináris operátorokból (+,-,*,/) áll. Készítsünk egy olyan programot, ami a kifejezést fordított lengyel formára hozza!

I.117. $*$ Adott egy kifejezés, ami egybetűs vátozókból, zárójelekből és bináris operátorokból (+,-,*,/) áll. Készítsünk egy olyan programot, ami kiszámítja a kifejezéshez az alábbiakban definiált az $S(K)$ értéket! Írjuk fel a kifejezés szintaxisfáját, jelölje a fát K, a baloldali részfát K_l, a jobboldalit K_r. Ha K az üres fa, akkor $S(K) = 0$, különben pedig a következő rekurzív formulával adjuk meg az értékét:

$$S(K) = \begin{cases} S(K_l) + 1 & , \textit{ha } K_l = K_r \\ max(S(K_l), S(K_r)) & , \text{ különben} \end{cases}$$

I.118. $*$ Adott egy kifejezés, ami egybetűs vátozókból, zárójelekből és bináris operátorokból (+,-,*,/) áll. Készítsünk egy olyan programot, ami kiszámítja a kifejezés szintaxisfájának magasságát!

I.119. ∗ Adott egy kifejezés szintaxisfájának preorder és inorder bejárása során kapott sorozat. Készítsünk programot, ami ebből előállítja a szintaxisfa postorder bejárásakor előálló szimbólumsorozatot!

I.120. ∗ Készítsünk olyan programot, ami beolvas egy logikai kifejezést, és kinyomtatja az igazságtábláját! A kifejezés egybetűs változókat, logikai operátorokat és (kerek) zárójeleket tartalmazhat. (Egyszerűbb a feladat, ha feltesszük, hogy a kifejezés szintaktikusan helyes.)

I.121. Adott egy szövegfile, amelynek a sorai nem haladják meg a 80 karaktert. Másoljuk át a file-t úgy, hogy a szavak közé szóközöket beszúrva minden sort kiegészítünk 80 hosszúságúra!

I.122. Adott egy file-ban egy szöveg, ami csak betűkből áll. Nyomtassuk a szöveget úgy, hogy minden betűt Morze-jelekkel ábrázolunk! A Morze-jelek között hagyjunk egy-egy szóközt, és egy szót nyomtassunk egy sorba!

I.123. Készítsünk egy interaktív szótárprogramot! A program számára lehessen definiálni egy angol szó jelentését, és lehessen visszakérdezni egy szó jelentését! Visszakérdezéskor engedjük meg, hogy a kérdezett szavakat rövidítsék! A definiált szavakat tároljuk lemezen! (A gyors visszakeresés érdekében használhatunk pl. kiegyensúlyozott bináris fát.)

I.124. Állapítsuk meg két sztringről, hogy azok a szóközöktől eltekintve megegyeznek-e!

I.125. Adva van egy távirat szövege. Készítsünk programot, ami meghatározza a távirat továbbításáért fizetendő díjat!

I.126. Adva van egy névsor, amiben a nevek egy vezetéknévből, és valahány utónévből állnak. Készítsünk programot, ami úgy írja ki a neveket, hogy a vezetéknévnek csak a kezdőbetűje legyen nagy betű, a vezetéknév betűi között legyen egy-egy szóköz, majd utána egy vessző, s ezután csak az utónevek nagy kezdőbetűi álljanak, mindegyik ponttal lezárva!

I.127. Tegyük fel, hogy horizontális tabulátor karaktereket tartalmazó szöveget akarunk listázni olyan nyomtatón, ami nem ismeri a tabulátorokat. Készítsünk egy programot, ami előkészíti a szöveget a nyomtatáshoz: a tabulátor karaktereket a megfelelő számú szóközzel helyettesíti! Tegyük fel, hogy a tabulátor-pozíciók az 1, 9, 17, ... pozíciók. (Általánosabb lesz a megoldásunk, ha a tabulátor-pozíciók felsorolásával paraméterezhető a programunk.)

I.128. Az előző feladat „megfordítása" is értelmes: Készíthetünk olyan programot, amely egy szöveg-file-ban a hosszú szóköz-sorozatokat úgy rövidíti le, hogy tabulátor karakterekkel cserél fel valahány szóközt.

I.129. Készítsünk olyan csomagot, ami eljárásokat exportál egy infix formájú kifejezés postfix alakjának előállítására (a kifejezésben csak az aritmetikai alapműveletek jelei, valós számok és zárójelek szerepelnek), és egy postfix formájú kifejezés értékének kiszámítására!

I.130. $*$ Készítsünk egy nem interaktív szövegszerkesztő programot! A program egy file-ból beolvasott parancsok alapján módosítsa a text-file-t úgy, hogy az eredmény egy új file-ban keletkezzen. A módosító parancsok a következők (a módosító parancsokban a sorok sorszámait növekvő sorrendben kell megadni):
- #DEL *sorszám* { , *sorszám* } — törli a sorszámaikkal megadott sorokat;
- #INC *sorszám* — a megadott sorszámú sor után beszúrja a parancssor után a következő parancssorig megadott sorokat.

I.131. $*$ Készítsünk egy olyan programot, ami szintezve (tehát a beágyazottságuknak megfelelően) kiírja egy Pascal program összes eljárásának nevét!

I.132. Másoljunk át úgy egy szintaktikusan helyes Modula-2 modult, hogy a megjegyzéseket közben kihagyjuk belőle!

I.133. Határozzuk meg, hogy egy bizonyos szó a Modula-2 forrásprogram mely soraiban fordul elő!

I.134. Számoljuk meg egy szintaktikusan helyes Modula-2 implementációs modulban, hogy melyik alapszó hányszor szerepel!

I.135. Gyűjtsük ki egy szintaktikusan helyes Modula-2 implementációs modulból a benne definiált eljárások és függvények neveit!

I.136. Egy szintaktikusan helyes Pascal programban az alapszavak betűit változtassuk nagybetűkre, a megjegyzések és szövegliterálok betűit ne változtassuk, a többi betűt cseréljük kisbetűre!

I.137. Készítsünk olyan programot, ami egy szintaktikusan helyes Ada programban megkeresi azokat a címkéket, amelyekre nincsen hivatkozás!

I.138. Készítsünk programot, ami
a) elemez egy karaktersorozatot, s eldönti, hogy az egy szabályos Ada83 <*numeric literal*>-e!
b) megadja egy szabályos Ada83 <*numeric literal*> decimális értékét!
c) egy szabályos Ada83 <*numeric literal*>-nak kirajzolja a szintaxisfáját!

<*numeric literal*> ::= <*decimal literal*>|<*based literal*>
<*decimal literal*> ::= <*integer*> [. <*integer*>] [<*exponent*>]

<integer> ::= <digit> { [<underline>] <digit>}
<exponent> ::= E *[*] <integer>|* E – *<integer>*
<based literal> ::= <base> # *<based integer>*
[. <based integer>] # *<exponent>*
<base> ::= <integer>
<based integer> ::= <extended digit> { [<underline>] <extended digit>
<extended digit> ::= <digit>|<letter>

I.139. Adott egy CDL-nyelvű szabály (programegység) törzsének (némileg leegyszerűsített) szintaxisa. Készítsünk programot, ami egy sztringről eldönti, hogy az megfelel-e ennek a szintaxisnak!

<szabály> ::= <csoport> ”.”
<csoport> ::= <alternatíva>|<alternatíva> ”;” <csoport>
<alternatíva> ::= <tag>|<tag> ”,” <alternatíva>
<tag> ::= ”+” | ”-” | ”” | ”*” <azonosító> |*
”(” [<címke>] <csoport> ”)”

I.140. Adott az RSX-11M operációs rendszer parancsainak (némileg leegyszerűsített) szintaxisa. Készítsünk programot, ami egy sztringről eldönti, hogy az megfelel-e ennek a szintaxisnak!

<parancs> ::= <parancskód> <paraméterek> ”=” <paraméterek>|
<parancskód> <paraméter>
<paraméterek> ::= <paraméter>|<paraméter> <paraméterek>
<paraméter> ::= [<egység>] [<felhasználó>] <file>
<egység> ::= ”DK” <szám> ”:” | ”LP” <szám> ”:”
<felhasználó> ::= ”[” <szám> ”,” <szám> ”]”
<file> ::= <azonosító> [<toldalék>]
<toldalék> ::= <file-kiegészítő> [<verzió>]
<file-kiegészítő> ::= ”.” | ”.” <azonosító>| ”.” ””*
<verzió> ::= ”;” | ”;” ”” | ”.” ”*”*

I.141. Készítsünk egy telefonos információkat nyilvántartó programot! A program adjon lehetőséget a név és a telefonszám szerinti visszakeresésre is!

I.142. * Készítsünk egy órarendet nyilvántartó programot!
a) A program tudjon választ adni bizonyos kérdésekre, pl. meg tudja adni egy tanár óráinak időpontját és helyét, vagy azt, hogy egy adott teremben, adott időben kinek van órája stb.!
b) A program tudja ellenőrizni azt, hogy az órarendben nincsen-e ellentmondás!

I.143. Készítsünk programot könyvtári kölcsönzések nyilvántartására és a nyilvántartás lekérdezésére!

I.144. ** Készítsünk egy olyan programot, ami egy vasúti menetrendet tárol, s lehetőséget ad annak lekérdezésére! (Bonyolultabb a feladat, ha a program az átszállással elérhető állomásokra is meg tudja adni az indulási és érkezési időket.)

I.145. Adott három file, ami egy öttusa-verseny eredményeit tartalmazza. Az első file rekordjai az indulók rajtszámait és nevét tartalmazzák. A második file a csapatok nevét és a csapatok három versenyzőjének a rajtszámát tartalmazzák. A harmadik file az elért eredményeket tartalmazza olymódon, hogy egy rekord a versenyző rajtszámából, a versenyszám megnevezéséből és az elért pontszámból áll. Készítsünk táblázatot az egyéni versenyről és a csapatok versenyéről!

I.146. A metrón utasszámlálást végeznek: az állomásokat megsorszámozzák, s minden állomáson minden utasnak a kezébe adnak egy kártyát az állomás sorszámával, a kiszálló utasoktól pedig a kártyákat összegyűjtik. Az összegyűjtött kártyákat minden óra végén egy-egy zsákba zárják. A kártyák megszámlálása után ismerjük minden állomásra minden óra végén az egyes állomásokról érkezett utasok számát. Készítsünk olyan programot, ami különféle grafikonok nyomtatásával segíti az adatok kiértékelését!

I.147. Készítsünk egy tetszőleges elemekből álló halmazt megvalósító csomagot (a halmazt reprezentálhatjuk pl. egy hash-táblával)!

I.148. Készítsünk egy tetszőleges elemekből álló kétirányú listát megvalósító csomagot (Create, PutFirst, Put, PutLast, GetFirst, Get, GetNext, GetPrev, Overwrite, Seek, Delete, Concat, Isempty, Length műveletekkel)!

I.149. Készítsünk egy tetszőleges elemekből álló rendező-fát megvalósító csomagot (create, isempty, write, first, read műveletekkel)!

I.150. Készítsünk egy tetszőleges elemekből álló n-áris fát megvalósító csomagot!

I.151. Készítsünk olyan programot, ami eldönti egy irányítatlan egyszerű gráfról, hogy van-e olyan köre, ami minden csúcson áthalad!

I.152. Készítsünk olyan programot, ami eldönti egy irányítatlan egyszerű gráfról, hogy van-e olyan csúcsa, amit minden más csúccsal él köt össze!

I.153. Készítsünk olyan programot, ami eldönti egy irányítatlan egyszerű gráfról, hogy az teljes gráf-e (vagyis, hogy bármely két csúcsa össze van-e kötve)!

I.154. Készítsünk olyan programot, ami meghatározza egy irányítatlan egyszerű gráf komplementer gráfját (azaz: megadja, hogy milyen éleket kellene hozzáadni a gráfhoz, hogy teljes gráfot kapjunk)!

I.155. ** Készítsünk programot, ami meghatározza egy irányítatlan gráf összes teljes részgráfját!

I.156. * Készítsünk programot, ami egy irányított gráfban megkeresi azt a csúcsot, amelyből minden más csúcs elérhető (úttal)!

I.157. * Valósítsunk meg privát típusként egy irányított gráf típust!

I.158. * Készítsünk programot, ami egy irányított gráfban, ahol az élekhez hosszértékeket is rendeltünk, megkeresi egy adott csúcsból a többi csúcsba vezető legrövidebb utakat!

I.159. ** Készítsünk olyan programot, ami eldönti egy irányítatlan egyszerű gráfról, hogy hányszorosan összefüggő!

I.160. * Készítsünk olyan programot, ami egy gráfban keres egy olyan utat a p és q csúcsok között, ami nem halad át az x csúcson!

I.161. * Készítsünk olyan programot, ami egy gráfban meghatároz egy olyan utat, ami minden élet pontosan egyszer tartalmaz!

I.162. * Készítsünk olyan programot, ami megkeresi egy irányítatlan egyszerű gráfban a minimális kört!

I.163. ** Készítsünk olyan programot, aminek adata két irányítatlan egyszerű gráf, s eldönti, hogy az első részgráfja-e a másodiknak!

I.164. ** Adott egy irányítatlan, de esetleg többszörös éleket tartalmazó síkbarajzolható gráf, ami egy kémiai szerkezet rajza. Készítsünk olyan programot, ami megkeresi azokat az atomokat, amelyek több gyűrűben is benne vannak!

I.165. Adott egy irányítatlan, de esetleg többszörös éleket tartalmazó síkbarajzolható gráf. Ez egy kémiai szerkezet rajza, ami csak a szénatomokat tartalmazza. Készítsünk olyan programot, ami meghatározza, hogy az egyes szénatomokhoz hány (a rajzon fel nem tüntetett) hidrogén kapcsolódik!

I.166. ** Adott egy irányítatlan, de esetleg többszörös éleket tartalmazó síkbarajzolható gráf, ami egy kémiai szerkezet rajza. Készítsünk olyan programot, ami megszámolja, hogy a szerkezet hány gyűrűt tartalmaz!

I.167. $*$ Készítsünk programot, ami meghatározza egy irányítatlan gráf egy feszítőfáját!

I.168. $*$ Adva van N darab elvégzendő feladat, s egy táblázat, amelyben feltüntettük, hogy melyik feladat elvégzése mennyi ideig tart, s melyek azok a feladatok, amiket csak a feladat befejezése után lehet elkezdeni. Készítsünk olyan programot, ami meghatározza, hogy az összes feladat elvégzése legkevesebb mennyi időt igényel!

I.169. Egy csoport elemeit az 1, a, b jelekkel ábrázoljuk, s adott a csoport szorzási táblája. Készítsünk programot, ami kiszámítja egy szorzat (az 1, a, b jelekből álló sorozat) eredményét!

$*$	1	a	b
1	1	a	b
a	a	b	1
b	b	1	a

I.170. $**$ Adott egy csoport a szorzási táblájával. Készítsünk programot, ami kiszámítja a csoport generátorát (azaz: az csoport elemeinek azt a részhalmazát, amelynek elemeiből szorzással előállítható a csoport többi eleme)!

I.171. Adott a H és G csoport a szorzási táblájával. Készítsünk programot, ami eldönti, hogy H részcsoportja-e a G csoportnak (azaz: az H elemei egyben G elemei, s a szorzás eredménye H-ban ugyanaz, mint G-ben)!

I.172. $**$ Adott a H és a G csoport a szorzási táblájával. Készítsünk programot, ami eldönti, hogy a csoportok izomorfak-e (azaz: található-e egy-egy értelmű művelettartó megfeleltetés a H és a G elemei között)!

I.173. Adott egy játéktábla, amelyen egymás mellett nyolc lyuk van, mindegyikbe legfeljebb egy golyó fér. Kezdetben a baloldali négy lyukban egy-egy fekete, a jobboldali háromban egy-egy fehér golyó van. A megengedett műveletek: egy golyót áthelyezhetünk a szomszédos üres lyukba, vagy pedig egy golyót átugorva a második lyukba (ha az üres). A fekete golyókat azonban csak jobbra, a fehéreket csak balra vihetjük. Készítsünk programot, ami a fenti szabályokat betartva átmozgatja a fekete golyókat a tábla jobboldalára, s fehér golyókat a tábla baloldalára!

I.174. Adott egy játéktábla, amelyen egymás mellett kilenc lyuk van, s mindegyikben egy-egy fekete vagy fehér golyó van. Készítsünk programot, ami a szomszédos golyók felcserélgetésével átmozgatja a fekete golyókat a tábla jobboldalára, a fehér golyókat pedig a tábla baloldalára!

I.175. Adott egy játéktábla, amelyen egymás mellett kilenc lyuk van, s mindegyikben egy-egy kék, fehér vagy piros golyó van. Készítsünk programot, ami rendre tetszőleges két golyót felcserélve átmozgatja a kék golyókat a tábla baloldalára, a piros golyókat a tábla jobboldalára, míg a fehér golyók középen maradnak!

I.176. Adott két halom gyufaszál. Két játékos felváltva lép, s az nyer, aki az utolsó gyufaszálat felveszi! Egy lépés abból áll, hogy a játékos az egyik halomból elvesz tetszőleges számú (de legalább egy) gyufaszálat. Készítsünk interaktív programot, ami eljátsza az egyik játékos szerepét!

I.177. Adott két halom gyufaszál. Két játékos felváltva lép, s az nyer, aki az utolsó gyufaszálat felveszi! Egy lépés abból áll, hogy a játékos az egyik halomból elvesz tetszőleges számú (de legalább egy) gyufaszálat, vagy pedig mindkét halomból elvesz ugyanannyi (de legalább egy) gyufaszálat. Készítsünk interaktív programot, ami eljátsza az egyik játékos szerepét!

I.178. $*$ Készítsünk olyan programot, ami megállapítja, hogy egy üres sakktáblán minimálisan hány lépésre van szüksége egy huszárnak, hogy egy adott A mezőről egy adott B mezőre jusson!

I.179. Készítsünk programot, ami meghatározza 12 huszárnak minden olyan elhelyezését a sakktáblán, amelyben minden mezőt támadnak!

I.180. Készítsünk egy rekurzív eljárást 8 vezér elhelyezésére egy sakktáblán úgy, hogy egyik se üsse a másikat!

I.181. $*$ Készítsünk olyan programot, ami megállapítja, hogy a sakktáblán egy adott állásban
a) világos tehet-e gyaloglépést,
b) világos képes-e ütni valamelyik gyalogjával,
c) világos képes-e sakkot adni!

I.182. $*$ Készítsünk olyan programot, ami megállapítja, hogy a sakktáblán egy adott állásban világos hány lépést tehet!

I.183. $**$ Készítsünk olyan sakkprogramot, ami
a) a világos királlyal és a vezérrel játszik a sötét király ellen;
b) a világos királlyal és két bástyával játszik a sötét király ellen!

I.184. $**$ Készítsünk olyan programot, ami megállapítja, hogy a sakktáblán egy adott állásban
a) világos adhat-e mattot egy lépésben,
b) világos adhat-e mattot n lépésben!

I.185. Készítsünk interaktív programot, ami segít a történelmi dátumok tanulásában! A program tegyen fel kérdéseket (pl. "Mikor volt a mohácsi csata?"), s adja meg, hogy a válasz helyes volt, vagy sem. Rossz válasz esetén adja meg a helyes évszámot is, s néhány kérdés múlva tegye fel ismét ugyanezt a kérdést.

I.186. Készítsünk interaktív programot, ami segít valamely idegen nyelv szavainak tanulásában! A program kérdézze meg egy szó jelentését, s adja meg, hogy a válasz helyes volt, vagy sem. Rossz válasz esetén adja meg a helyes szót is, s néhány kérdés múlva tegye fel ismét ugyanezt a kérdést.

I.187. Válasszunk ki egy olyan helyzetet, amelyben N személy hasonló tevékenységet végez, s a tevékenység megkezdése és folytatása valami feltételtől függ. Ezután készítsünk tasko(ka)t a folyamatokat szimulálására, s helyezzünk el a programban megfelelő kiírásokat a folyamat nyomon követhetősége érdekében! Az egyes személyek paramétereit változtatva érdekes megfigyelni, hogyan változik az egész rendszer működése. Ilyen tevékenység lehet pl. a sörözés, ahol a folytatás feltétele az, hogy a korsóban legyen még sör. Ebben a feladatban nyilván szükség van egy kocsmárosra is, aki időnként körbejár, s tölt az üres korsókba.

I.188. Készítsünk programot egy nemdeterminisztikus automata működésének szimulálására taskok segítségével olymódon, hogy nem egyértelmű esetben a program indítson el egy újabb taskot! Az automata legyen

$$A = (\{q0, q1, q2, q3, q4, q5\}, \{a, b, c\}, \delta, q0, \{q4\}),$$

ahol

$$\begin{aligned}
\delta(q0, a) &= \{q0, q1\}, \\
\delta(q0, b) &= \{q0, q2\}, \\
\delta(q0, c) &= \{q4\}, \\
\delta(q1, c) &= \{q0, q3\}, \\
\delta(q2, b) &= \{q2\}, \\
\delta(q2, c) &= \{q0, q3\}, \\
\delta(q3, c) &= \{q4\},
\end{aligned}$$

az összes többi esetre $\delta(x, y) = \{q5\}$.

I.189. Készítsünk általános eljárást egy determinisztikus automata működésének szimulálására! Az állapotok halmaza (típusa), a bemenő jelek halmaza (típusa), az átmeneti függvény, a kezdő- és végállapotok mind legyenek az eljárás paraméterei!

I.190. Adva van két edény: az egyikben az $1, 2, \dots, n$ számokkal megjelölt golyók vannak, a másik üres. Állítsunk elő 1 és n közötti egyenletes eloszlású véletlenszámokat, s az adott számmal jelölt golyót mindig tegyük át a másik edénybe. Készítsünk programot, ami szimulálja ezt a folyamatot, s vizsgáljuk az edények tartalmának változását!

I.191. Egy részeg (R pont) ember áll egy lámpaoszlopnál (origó). Arra vagyunk kiváncsiak, hogy N lépés után milyen messzire jut az oszloptól. Minden lépése egységnyi hosszúságú, s az iránya véletlenszerű. Készítsünk programot, ami K db kísérletet végez, s ezek alapján megadja a várható távolságot!

I.192. * Egy zárt teremben 30 dohányos ember van. Minden időegység végén 20% annak a valószínűsége, hogy eggyel nő, 20%, hogy eggyel csökken az égő cigaretták száma, 60%, hogy nincs változás. Egy cigaretta egy időegység alatt egy egységnyi füstmennyiséget termel. Ha a teremben a füstmennyiség eléri a 200 egységet, akkor a számítógép elindítja az elszívóberendezést, amely egy időegység alatt 30 egységgel csökkenti a teremben a füstmennyiséget. Ha a teremben a füstmennyiség 40 egységnél kevesebb, akkor a gép kikapcsolja az elszívót. Készítsünk programot, ami időegységenként kiírja a füstmennyiség értékét, a dohányzók számát és az elszívó kapcsolójának állását!

I.193. A bűvös négyzet egy olyan négyzetes mátrix, amelynek sorösszegei, oszlopösszegei és a fő- ill. a mellékátlójában levő elemek összege megegyezik, s ez az összeg $\frac{n(n^2+1)}{2}$ (ahol n -nel jelöltük a négyzet oldalát, és n páratlan). Készítsünk programot, ami az n érték ismeretében elkészít egy bűvös négyzetet!

I.194. ** A latin négyzet egy olyan négyzetes mátrix, amelynek sorai és oszlopai az $1, \dots, n$ számok különböző permutációi (n -nel jelöltük a négyzet oldalát). Készítsünk programot, ami az n érték ismeretében elkészít egy latin négyzetet!

I.195. Készítsünk egy programot, ami egy számsorozatról megállapítja, hogy periodikus-e (egy sorozat akkor periodikus, ha előállítható egy elemsorozat többszöri egymás mellé írásával)!

I.196. Készítsünk programot, ami az 1,2,3,...,9 számok közé + és − jeleket helyez el úgy, hogy a kifejezés értéke egy előre megadott szám legyen egyenlő! (Pl. $122 = 12 + 34 - 5 - 6 + 78 + 9$.)

I.197. Készítsünk olyan programot, ami előállítja
a) az első n természetes szám összes permutációját!
b) az első n természetes szám összes olyan permutációját, ahol $p_i \neq i$ $(1 \leq i \leq n)$!

I.198. ** Egy vállalat N állást hirdet meg. Jelentkezik rá ugyanannyi pályázó, s mindegyik közli, hogy melyik állásokat tudná betölteni. Készítsünk programot, ami eldönti, hogy a pályázókkal betölthető-e minden állás, s ha igen, akkor adja meg, melyiket milyen állásra kell felvenni!

I.199. * Egy teherautó A -ból B -be megy. Az út mentén vannak az A_i, B_i helyek tetszőleges sorrendben (de a megfelelő B_i mindig az A_i párja után van). Minden A_i helyről a hozzá tartozó B_i helyre kell szállítani egységnyi mennyiséget úgy, hogy a teherautó egyszerre csak egy terhet szállíthat. Készítsünk programot, ami eldönti, hogy melyik szállításokat kell elvállalni ahhoz, hogy a legtöbb igényt kielégítsük!

I.200. Adott N darab tárgynak a súlya. Osszuk a tárgyakat két csoportra úgy, hogy a két csoport súlya a lehető legközelebb legyen egymáshoz!

J. Függelék

Szótár

abstract absztrakt
access mutató
alternative ág (case)
ancestor ős
anonymous névtelen
array tömb
assignment értékadás
barrier őrfeltétel
based literal adott alapú számrendszerben adott számliterál
block statement blokk utasítás
body törzs
bounded error korlátozott hiba
busy waiting aktív várakozás
callable entity hívható egyed, hívható egység
calling convention hívási konvenció
character literal karakterliterál
choice parameter kiválasztó paraméter
classwide type osztályszintű típus
clause záradék
comment megjegyzés
compilation order fordítási sorrend
compilation unit fordítási egység
completion megvalósítás, befejezés
compound statement összetett utasítás
condition feltétel, logikai kifejezés
conformance (full/mode/subtype) teljes/mód/altípus kompatibilitás
controlled type ellenőrzött típus

deadlock holtpont
decimal literal decimális formában adott számliterál
declarative region deklarációs környezet
default value alapértelmezett vagy feltételezett érték
deferred constant késleltetett konstans
delimiter elhatároló jel
designated megjelölt
digit számjegy
direct visibility direkt láthatóság
distributed system (el)osztott rendszer
encapsulation egységbezárás, tokbazárás
entity egyed, entitás
entry belépési pont
entry family belépésipont-család
entry queue várakozási sor
enumeration type felsorolási típus
exception kivétel
expanded name kiterjesztett név
exponent kitevő (exponens)
expression kifejezés
extended scope kiterjesztett hatáskör
extended visibility kiterjesztett láthatóság
full view teljes nézet
function függvény
general access type általános mutató típus
generic sablon
global globális
graphic character grafikus karakter
guard őrfeltétel
handler kezelő
hide eltakar
identifier character azonosítóbetű
immediate scope közvetlen hatáskör
inheritance öröklődés
instance példány
instantiation példányosítás
interrupt megszakítás
label címke

lexical element lexikális elem
library element könyvtári elem
library package könyvtári csomag
library procedure könyvtári eljárás
library unit könyvtári egység
lifetime élettartam
limited korlátozott
local lokális
lock lakat
loop ciklus
loop name ciklus neve
mutual exclusion kölcsönös kizárás
named association névvel jelölt paraméter-megfeleltetés
namespace címtér
obsolete elavult
occurence előfordulás, példány
overloading átlapolás
overriding felüldefiniálás
package csomag
parent szülő
partial view részleges nézet
partition partíció
pass by copy másolat szerinti paraméterátadás
pass by reference hivatkozás szerinti paraméterátadás
predefined előre definiált
private child titkos gyermek
private descendant titkos leszármazott
private part átlátszatlan rész
private type átlátszatlan típus
procedure eljárás
program library programkönyvtár
protected védett
public child nyilvános gyermek
public descendant nyilvános leszármazott
real-time system valós idejű rendszer
renaming átnevezés
reserved word kulcsszó
return-by-copy másolat visszaadás

return-by-reference hivatkozás visszaadás
root gyökér
runtime error futási idejű hiba
scope hatáskör
semantics dependencies szemantikus függőségi viszonyok
separator elválasztó jel
single protected object egyedüli védett objektum
simple statement egyszerű utasítás
special character speciális karakter
specification specifikációs rész
statement utasítás
static expression statikus kifejezés
stream adatfolyam
string literal sztring literál
stub csonk
subprogram alprogram
subtype altípus
subunit alegység
tagged jelölt
time-sharing system időosztásos rendszer
type extension típuskiterjesztés
unconstrained array megszorítás nélküli indexhatárú tömb
use clause use utasítás
view nézet
visibility láthatóság
visibility rules láthatósági szabályok
visible part látható rész
with clause with utasítás

Irodalomjegyzék

[1] *Ada Reference Manual*,
http://www.ada95.com

[2] *Annotated Ada Reference Manual*,
http://www.ada95.com

[3] *Ada 95 Rationale*,
http://www.ada95.com

[4] Barnes, J.: *Programming in Ada 95*,
2nd edition, International Computer Science Series, Addison-Wesley Longman Limited, 1998.

[5] Burns, A. - Lister, A. M. - Wellings, A. J.: *A Review of Ada Tasking*,
Lecture Notes in Computer Science 262,
Springer-Verlag, 1987.

[6] Cohen, N. H.: *Ada as a second language*,
second edition, McGraw-Hill, 1996.

[7] Fóthi, Á.: *Bevezetés a programozáshoz*,
ELTE TTK, Budapest, 1983.

[8] Gehani, N. H.: *Ada Concurrent Programming*,
Prentice-Hall, 1984.

[9] Habermann, N. A.: *ADA for Experienced Programmers*,
Addison-Wesley, 1983.

[10] Horowitz, E.: *Magasszintű programnyelvek*,
Műszaki Könyvkiadó, Budapest, 1987.

[11] Horváth, Z.: *Az Ada taszkokról bővebben*
(12 oldal), készült Sheffieldben, 1992. júl.-aug.
ftp://valerie.inf.elte.hu/pub/parh/ada.ps

[12] Kozics, S.: *Az Ada programozási nyelv*,
ELTE, Budapest, 1992.

[13] Kozics, S.: *Az Algol60, a FORTRAN, a COBOL és a PL/I programozási nyelvek*,
ELTE, Budapest, 1992.

[14] Kozsik, T.:*Ada tanfolyam*,
http://www.elte.hu/ kto/adagyak/

[15] Liskov, B.H., Guttag, J.: *Abstraction and Specification in Program Development*,
M.I.T. Press, Cambridge (Mass.), 1986.

[16] Marcotty, M. - Ledgard H.: *The World of Programming Languages*,
Springer, New York-Berlin-Heidelberg, 1987.

[17] Meyer, B.: *Object-Oriented Software Construction*,
Prentice Hall, New York, 1988.

[18] Nyékyné Gaizler, J. et al.: *Java 1.1 útikalauz programozóknak*,
ELTE, TTK, Hallgatói Alapítvány, 1998.

[19] Olsen, E. W. - Whitehill, S. B.: *Ada for Programmers*,
Reston Publishing Co., Prentice-Hall, 1983

[20] Pyle, I. C.: *Az Ada programozási nyelv*,
Műszaki Könyvkiadó, Budapest, 1987.

[21] Sethi, R.: *Programming languages*,
Addison-Wesley, 1996.

[22] Smith, M. A.: *Object-oriented Software in Ada95*,
Thomson, 1996.

[23] *Turbo Pascal 5.5 - Object-Oriented Programming Guide*,
Borland, 1989.

[24] U.S. Department of Defense: *The Programming Language Ada, Reference Manual*,
American National Standards Institute, Inc. ANSI/MIL-STD-1815A-1983,
Lecture Notes in Computer Science 155, Springer-Verlag, 1983.

[25] Watt, D. A. - Wichmann, B. A. - Frindlay, W. : *ADA Language and Methodology*,
Prentice-Hall International, Series in Computer Science 1987

Tárgymutató

A

abort utasítás 309, 318, *321*
abs operátor 52, 55
abstract kulcsszó 240
absztrakt
 alprogram 115
 típus 41, 228, 240, 261
accept utasítás 159, 164, 309, 315–317, 319, 321, 323
 kivételkezelő része *127*
access kulcsszó 246
access típus 77
Ada csomag 377
Ada.Exceptions csomag 133, 257
Ada.Finalization csomag 158, 290
Ada.Integer_Text_IO csomag 253
Ada.Interrupts.Names csomag 354
Ada.Interrupts csomag 349
Ada.Streams csomag 303
adatfolyam 303
 attribútum 304
 'Class'Input 305
 'Class'Output 305
 'Class'Read 305
 'Class'Write 305
 'Input 305
 'Output 305
 'Read 304
 'Write 304
Adjust eljárás
 → típus ellenőrzött
aggregátum
 rekord → rekord
 tömb → tömb
aktív várakozás 345
aktuális paraméter 42, 92, 102
 sablonhoz 227
alapértelmezett érték → feltételezett érték
alapértelmezett mód 107, 235
alegység 164, 173, *175*, 347
algoritmus absztrakció 228
aliased kulcsszó 111, 148, 246
Alignment → sztringkezelés
all kulcsszó 247
Allocate 369
allokálás *523*
allokátor 146, 158
alprogram 35, 38, 41, 49, 77, 83, 88, 92, 96, *101*, 144, 145, 158, 159, 163, 165, 167, 173, 312, 313, 315, 317
 átnevezés *113*
 deklaráció *113*
 egymásba ágyazott *115*
 formális paraméter 35
 globális változója *103*
 kivételkezelő része *127*
 lokális változója *103*
 megvalósítás 115
 mint sablon paramétere 228
 öröklött 150, 152
 paraméter *102*
 módja *105*

sablon 166, *228*
specifikáció *113*
törzs 113, 166
altípus *38*, 39–41, 46, 91, 94, 168
deklaráció 39
kompatibilitás 116
`and` operátor 51
`and then` operátor 52, 119
`Append` 398, 399, 407
`Arccos` → numerikus csomag
`Arccosh` → numerikus csomag
`Arccot` → numerikus csomag
`Arccoth` → numerikus csomag
`Arcsin` → numerikus csomag
`Arcsinh` → numerikus csomag
`Arctan` → numerikus csomag
`Arctanh` → numerikus csomag
`array` 61, 244
`ASCII` csomag 381
Assembly 365
`Asynchronous_Task_Control` csomag 483
aszinkron
eljáráshívás 491
programegységek 309
`Atomic_Components` → pragma
`Atomic` → pragma
atomi
objektum 469
típus 469
`Attach_Handler` eljárás 349
`Attach_Handler` pragma 349
attribútum 40, *46*, 48, 50, 51, 54, 57, 58, 117, 146, 248
`'Access` *77*, 79, 80, 83, 85, 88, 111
`'Adjacent` 423
`'Address` 357
`'Aft` 58
`'Alignment` 357
`'Base` *46*
`'Bit_Order` 360
`'Component_Size` 358
`'Compose` 422
`'Copy_Sign` 424
`'Count` 351
`'Delta` 58
`'Digits` 57, 58
`'Exponent` 422
`'External_Tag` 358
`'First` *46*
`'First_Bit` 360
`'Floor` 422
`'Fore` 58
`'Fraction` 422
`'Identity` 133, 257
`'Last` *46*
`'Last_Bit` 360
`'Image` *47*
`'Leading_Part` 424
`'Machine` 424
`'Machine_Emax` 421
`'Machine_Emin` 421
`'Machine_Mantissa` 421
`'Machine_Overflows` 421, 425
`'Machine_Radix` 420, 424
`'Machine_Rounds` 421, 425
`'Max` *46*
`'Min` *46*
`'Modulus` 55
`'Partition_ID` 486
`'Pos` *48*
`'Position` 360
`'Pred` *47*
`'Range` *46*
`'Remainder` 423
`'Round` 58
`'Rounding` 423
`'Scale` 58
`'Scaling` 422
`'Signed_Zeros` 422
`'Size` 357
`'Small` 58
`'Succ` *47*
`'Storage_Pool` 369
`'Storage_Size` 358, 369
`'Tag` 275, 278

'Unbiased_Rounding 423
'Unchecked_Access 83
'Val *48*
'Value *48*
'Wide_Image *47*
'Wide_Value *48*
'Wide_Width *47*
'Width *48*
Access 111
adatfolyam 304
'Class'Input 305
'Class'Output 305
'Class'Read 305
'Class'Write 305
'Input 305
'Output 305
'Read 304
'Write 304
tömb 64
'First 65
'First(N) 65
'Last 65
'Last(N) 65
'Length 65
'Length(N) 65
'Range 65
'Range(N) 65
'Unchecked_Access 368
'Valid 367
attribútumdefiníciós záradék 356
azonnali
objektum 469
típus 470
azonosító 31, 35, 36, 38, 42, 49, 163, 166
elsődleges 163
globális *522*
lokális *522*
másodlagos 163
azonosítóbetű *31*

Á

átlapolás 36, 49, 104, 117, *167*, *523*
átlátszatlan rész 164, 339
átlátszatlan típus *146*, 160, 235
átlátszatlan típuskiterjesztés 146, 152
átnevezés 113, 115, 167, *169*, 347, 348
mint deklaráció 115
mint törzs 116

B

bázistípus *46*
beépített típus 36
belépési pont 115, 146, 159, 309, 310, *315*, 316, 319, 320, 321, 323, 324
hozzárendelt törzs 335
kivételkezelő része *127*
őrfeltétele 335
taszkban 333
várakozási sora 335
védett objektumban *334*
belépésipont-család 318, 337
bitsorrend 360
biztonságos számok *56*
blokk 44, 45, 49, 77, *96*, 159, 163, 166, 521
declare utasítás *96*
kivételkezelő része *127*
törzs 97
blokkstruktúra *522*
Boolean típus 51

C

C 443, 446
case utasítás → elágazás
Ceiling 423
Character_Mapping_Function → stringkezelés

Character_Range
→ stringkezelés
Characters csomag 383
Characters.Handling csomag 383
Characters.Latin_1 csomag 383
Character_Set → sztringkezelés
Character típus 51
ciklus *94*, 164
 exit utasítás 95, *97*, 137
 fej 95
 feltétel 95
 feltétel nélkül 95
 for utasítás *95*
 in kulcsszó *94*, 337
 loop kulcsszó 95
 mag 95, 97
 reverse kulcsszó *95*
 változó 42, 94
 while utasítás *95*
címtér 171
Clear → SVGALib
Close
→ Sequential_IO,
→ Direct_IO,
→ Text_IO
→ Stream_IO
COBOL 443, 455
Col → Text_IO
Command_Line csomag *440*
Communications_Error 491
Complex_IO csomag 513, 515
constant kulcsszó 37, 247
Constraint_Error kivétel 38, 43, 47, 48, 50, 56, 91, 126
Controlled típus 286, 290, 298
Convention → pragma
Cos → numerikus csomag
Cosh → numerikus csomag
Cot → numerikus csomag
Coth → numerikus csomag
Count → sztringkezelés
Create → Sequential_IO,
→ Direct_IO,
→ Text_IO
→ Stream_IO
Current_Error → Text_IO
Current_Input → Text_IO
Current_Output → Text_IO

Cs

csomag 36, 38, 45, 51, 54, *143*, 159, 168, 173, 250, 251
 átlátszatlan rész 143
 private 144, 146
 beágyazott 145
 gyermek 143
 inicializáló rész 144
 kivételkezelő része *127*
 könyvtári 143
 látható rész 143
 mint sablon paramétere 228
 sablon 146, 152, 160, *228*
 specifikációs rész 38, *143*, 165
 törzs 143, *144*
 implicit 145
csonk 164, *175*

D

Data_Error kivétel 140, 221
Deallocate 369
Decimal_Conversions csomag 457
Decimal csomag 497
decimális osztás 497
decimális típus 244
declare utasítás → blokk
definit típus 238
deklaráció *35*, 38, 39, 41–44, 49, 50, 53, 143, 163
 átlapolás 165, *167*
 explicit 35
 hatáskör 35

implicit 35
kiértékelés 35
lokális 144
teljes 144
többszörös 43
deklarációs környezet 143, 160, *163*, 168
közvetlenül tartalmazó 164
deklarációs rész 77, 91, 144, 145
deklarálás 77, 79, 80
deklarált objektum 77
`delay` utasítás 309, *321*, 324
`Delete` → sztringkezelés, → `Sequential_IO`, → `Direct_IO`, → `Text_IO` → `Stream_IO`
`delta` kulcsszó 244, *57*, 58
`Denorm` 421
`Device_Error` 221
`digits` kulcsszó 244
dinamikus
kiválasztás 88
objektum 77, 80, 82
összekapcsolás 261
tartalmazás *130*
típus *77*
`Direct_IO` 425, *427*
`Direction` → sztringkezelés
diszkrét altípus 94
diszkrét típus 37, *48*, 94, 243
diszkrimináns 152, 153, 159, 237, 238, 311, 340
ismeretlen 238
jelölt típus 266
rekord típus → rekord
`Draw_Line` → `SVGALib`
`Draw_Pixel` → `SVGALib`
`Draw_Point` → `SVGALib`
`Duration` 382

E

egész
osztály 42
szám 35, 47, 55, 57, 59
típus 44, 48, *53*, 55
egyed 163, 166
egyedüli védett objektum *338*
egységbezárás 261, 350
ekvivalens típusok *36*
elágazás *93*, 124
`case` utasítás 73, *94*, 128
`others` ág 94
`when` ág 94
`else` kulcsszó 93
`elsif` kulcsszó 93
`if` utasítás *93*
`Element` 400, 408
elemi függvények → numerikus csomag
elemi típus *37*, 41
elhatároló jel 31, *33*
eljárás *101*
sablon 165, *228*
ellenőrzött típus → típus
elosztott program 485
előjel 55
előjeles egész típus 243
előre definiált könyvtárak
kivételei 127, 140
sablonjai 253
előre definiált művelet 146, 150
`else` kulcsszó → elágazás
`elsif` kulcsszó → elágazás
`End_Error` kivétel 140, 221
`end` kulcsszó 144
`End_Of_File` → `Sequential_IO`, → `Direct_IO`, → `Text_IO` → `Stream_IO`
`End_Of_Line` → `Text_IO`
`End_Of_Page` → `Text_IO`
entitás 35
definíció 35

név 35
nézet 35
entry hívási konvenció 348
entry kulcsszó 335
eredményintervallum 516
erős típusosság *36*, 37
Exception_Identity függvény 133
Exception_Id típus 133, 257
Exception_Information függvény 133
exception kulcsszó *127*, *128*
Exception_Name függvény 133
Exception_Occurance típus 132
exit utasítás → ciklus
Exp → numerikus csomag
explicit deklaráció 35
exponenciális alak 32
exponens *56*
Export → pragma
extremális elem 124

É

élettartam 144, *523*
értékadás 37, 39, 43, *91*, 158
értékhalmaz 38, 40, 42

F

fagyasztás 362
fájl
fájlobjektum *189*
hozzáférés
közvetlen *190*, 194
soros *190*, 193
külső *189*
megosztás *222*
szövegfájl *197*
oldalhossz *201*
sorhossz *201*
feldolgozási függőség 145
feldolgozó egység 485
felsorolási konstans 167
felsorolási típus 35, 37, 48, *49*, 50, 359
literál 116
feltételes kritikus szakasz 350
feltételezett érték 112
FIFO_Queuing 478
Finalize eljárás → típus ellenőrzött
Find_Token → sztringkezelés
Fine_Delta konstans 365
fixpontos
osztály 42
szám 47, 57
típus 37, *57*, 244
Flip → SVGALib
Flush → Text_IO
fordítás 36, 40, 44
fordítási egység 31, 114, 143, 163, 168, 229, 232, 347, *521*
fordítóprogram 38, *520*
for kulcsszó 337
Form 193, 218
formális paraméter 35, 42, 92, 102, 163, 165
sablonhoz 227
formázott output 513
forrásprogram *521*
forrásszöveg 31
Fortran 443, 461
for utasítás → ciklus
főprogram *103*
function kulcsszó *102*, 335
futási idejű hiba 125, 131
futási idő 35
függvény 38, 41, 43, 46, 48, 88, 92, *101*
hívás 79
mellékhatása *101*, 107, 159, 336, 340
név 35
sablon *228*
visszatérési értéke *102*, 105

G

generic kulcsszó *228*
Get → Text_IO
Get_Colours → SVGALib
Get_Immediate → Text_IO
Get_Line → Text_IO
Get_Palette → SVGALib
Get_Xdim → SVGALib
Get_Ydim → SVGALib
gépi alapszám 497
gépi kód 465
globális *164*,
 azonosító 522
 név 97
 változó 103, 124, 139, 339
GNAT fordító 266
goto utasítás *98*
grafikus karakter 31

Gy

gyermek
 csomag 143, 153, *160*, 163
 nyilvános 165
 titkos 160
gyökértípus 41, 42
gyűjtőtípus *77*, 81, 86

H

Has_Mode → SVGALib
határérték 54
hatáskör 143, 146, *163*, *164*, 168, *522*
 közvetlen 153, *164*
hatékonyság 40
Head 396, → sztringkezelés
helyesség *17*
hivatkozás 35
hivatkozás-visszaadás *108*, 347
hivatkozás szerinti paraméter-átadás *107*, 347
hívási konvenció 348
hívható egyed *165*
holtpont 342

I

időosztásos rendszer 350
if utasítás → elágazás
illesztés 91
Imaginary típus 506
implicit deklaráció 35, 150
Import → pragma
indefinit típus 238
Index → sztringkezelés,
 → Direct_IO
 → Stream_IO
Index_Non_Blank
 → sztringkezelés
infix alak 117, 118
inicializálás 91
Initialize eljárás
 → típus ellenőrzött
in kulcsszó → ciklus
in módú paraméter 105, 234
in operátor 119
in out módú paraméter 105, 234
input-output *189*, *425*
 adatfolyam *217*
 egész típusra *210*
 felsorolási típusra *215*
 valós típusra *212*
 Wide_Character *217*
 Wide_String *217*
Insert → sztringkezelés
Integer típus 35, 36, 39
Interfaces csomag 445
Interfaces.C.Pointers csomag 452
Interfaces.C.Strings csomag 450
Interfaces.C csomag 446
Interfaces.COBOL csomag 455
Interfaces.Fortran csomag 461
interpreter *520*

Interrupts csomag 467
Interrupts.Names csomag 468
intervallum 38, *46*, 49, 54, 56
 korlátozás 38
Intrinsic hívási konvenció 348
Invalid_Vga_Mode → SVGALib
IO_Exceptions csomag 425, *440*
Is_Alphanumeric 384
Is_Basic 384
Is_Character 385
Is_Control 383
Is_Decimal_Digit 384
Is_Digit 384
Is_Graphic 383
Is_Hexadecimal_Digit 384
Is_In 388
Is_ISO_646 385
is kulcsszó 113, 166
Is_Letter 383
Is_Lower 383
ISO-10646 szabvány 31
ISO-8859 szabvány 31
ISO-8859-1 szabvány 380, 386
Is_Open → Sequential_IO, → Direct_IO → Stream_IO
Is_Special 384
Is_String 385
Is_Subset 388
Is_Upper 383

J

jelölt
 rekord 37, 262
 típus 42, 150, 239
 diszkrimináns 266

K

karakter 47, 51, 58
 elhatároló jel *33*
 formázó 31
 grafikus 31
 kezelés *383*
 lánc 47
 literál 32, 35, 47, 51
 speciális 31
 típus *51*
karbantarthatóság *17*, 259
kategorizációs pragma 486
kevert forma *112*
kezdeti érték 36, 40, 42–45, 158
késleltetett konstans 160
kétszintes védelem 352
kiértékelés 145
kifejezés 43, 44, 46, 50–52, 59, 91, 102, 159
 statikus 94
kiterjesztett típus 241
kiválasztó paraméter 130, 132
kivétel 39, 91, 125, 257
 definiálása *127*
 előre definiált 125, 126, 127, 140
 kezelő 97, 164
 ág *128*
 rész *127*
 kiváltódása 125, *130*
 lekezelése 125
 példány 126, *132*
 programozó által definiált *127*
 programozó által kiváltott 125
 terjedése 125, *130*
kivételes esemény 159
kivételkezelt utasítássorozat 113, *127*
kódutasítás 366
kompatibilitás *18*
kompletté válás 159
komplett taszk 313, *313*, 315, 324
komplex
 aritmetika 505
 elemi függvények 510
 input-output 513
 konjugált 510
 számok 505

típusok 505
konstans 36, 38, 42, 44, 53, 54, 78, 85
késleltetett def. *148*
lokális 94
kontextus-váltás 334, 350
konvenció-azonosító 443
konverzió 41, 59, 159
érték 267
nézet 267
korlátozás 38–40, 54, 57
korlátozott 236
átlátszatlan típus 150, 236
nézet 150
típus 41, 91, 150, 158, 347
kölcsönös kizárás 334
könyvtári egység 159, 173, 347
gyökér 160, 173
sablon 229
szülő 160, 173
titkos 160
könyvtári elem 160, 173
környezet definíció 168
környezeti taszk 485
közvetlen fájl → fájl
kritikus szakasz 340
feltételes 350
kulcs 344
külső fájl → fájl

L

lakat 344
Latin-1 ábécé 31
`Layout_Error` 221
laza numerikus mód 505, 516
látható rész 168
láthatóság 143, *163*, *165*
direkt *165*, 168
eltakarás *166*
kiterjesztése 166
szelekciós *165*, 168
láthatósági kör *522*
lebegőpontos
szám 47, 56, 59
típus 37, *56*, 244
`Length` → sztringkezelés
leszármazott 42
nyilvános 160
titkos 160
lexikális elem *31*
lezárás 91
`limited` kulcsszó → korlátozott
`Limited_Controlled` típus 158, 286
`Line` → `Text_IO`
`Line_Length` → `Text_IO`
`Linker_Options` → pragma
Linux 223
`Lista` típus 289
literál 33, 38, 49, 164
`Log` → numerikus csomag
logikai kifejezés 93
logikai típus 41, *51*
lokalizált megjelenítés 498
ezredes vessző 499
lokál elemek 499
lokál karakterek 499
nulla elnyomó karakter 500
picture sztring 499
tizedes pont 499
valuta sztring 499
lokális *164*
alprogram 115
azonosító 97, 522
deklaráció 96
művelet 342
változó 103
`Look_Ahead` → `Text_IO`

M

makró 227
mantissza 56
`Max_Base_Digits` konstans 365
`Max_Binary_Modulus` konstans 365
`Max_Digits` konstans 365

Max_Int konstans 365
Max_Mantissa konstans 365
Max_Nonbinary_Modulus
konstans 365
másolat-visszaadás *108*
másolat szerinti
paraméterátadás *107*
megbízhatóság *17*
megjegyzés 33
megszakítás 349, 465
átadás 465
blokkolt 465
előfordulása 465
foglalt 465
függő 465
generálás 465
kezelő 465
megvalósítás 115, 348
mellékhatás → függvény mellékhatása
Membership → sztringkezelés
memóriagyűjtő 369
memóriatartalék 369
Memory_Size konstans 365
mester egyed 159
metódus 147
Min_Int konstans 365
minősítés 117, 337, 366
mód-kompatibilitás 116, 246, 247
Mode → Sequential_IO,
→ Direct_IO,
→ Text_IO
→ Stream_IO
Mode_Error 221
modellintervallum *56*, 516
modellorientált attribútum 517
modellszám 56, *56*, 516
mod kulcsszó 243
mod operátor 55
modul 521
moduló típus 53, 243
monitor 349
monoton idő 479
MouseButton → SVGALib
Mouse_Close → SVGALib
Mouse_Error → SVGALib
Mouse_Get_Button → SVGALib
Mouse_Get_Position
→ SVGALib
Mouse_Init → SVGALib
Mouse_Set_Position
→ SVGALib
MouseType → SVGALib
Mouse_Update → SVGALib
Mouse_Wait_For_Update
→ SVGALib
Move 390
mutató 38, 77–85, 87, 88
típus 77, 79–81, 83, 85, 86, 246
alprogramra mutató 227, 246, 348
mutató paraméter 110
művelet 58
absztrakt 282
átlapolás 277
primitív 266
rekord 68
tömb 64

N

Name → Sequential_IO,
→ Direct_IO,
→ Text_IO,
→ Stream_IO
Name_Error kivétel 140, 221
Natural típus 39
Neumann-elvű nyelvek 520
New_Line → Text_IO
New_Page → Text_IO
new utasítás 77, 240, 338
névtelen típus 80, 110, 310, 347
név túlterhelés → átlapolás
névvel jelölt forma *112*, 117, 231, 250
nézet *35*, 163
átlátszatlan rész *165*

látható rész *165*
részleges 146
teljes 146, 164
normalizált forma *56*
`not in` operátor 119
`not` operátor 51
`null` érték 77, 79, 84
`null` utasítás *92*
`Numeric_Error` kivétel 127
`Numerics` → numerikus csomag
`Numerics.Generic_Complex_Types` csomag 505
numerikus csomag *414*
elemi függvény *414*
komplex 510
véletlenszám *418*
nyilvános leszármazott 164, 168

O

objektum 35, 36, 41, 42, 45, 77–79, 81, 83, 84, 87, 260
deklaráció 36
deklarált 77
destruktor 286
dinamikus 77
elérési mód 35
értékadás 286
konstruktor 286
mint sablon paramétere 228
objektum-orientált
tervezés 259
oldalhossz → `Text_IO`
olvasók-és-írók probléma 335
`Open` → `Sequential_IO`, → `Direct_IO`, → `Text_IO`, → `Stream_IO`
operandus-intervallum 516
operátor 35, 38, 51, 52, 60, 117, 118, 230, 231, 248
felüldefiniálása 167
szimbólum 35
`or else` operátor 52, 119
`or` operátor 52
osztály 143, 147, 160, 260
osztályszintű
attribútum 160
típus 41, *42*, 238, 239
osztott rendszer 349, 485
`others` kulcsszó 129
`out` módú paraméter 105
`Overwrite` → sztringkezelés

Ö,Ő

öröklődés 139, 160, 167, 260, 262
összetett típus 37, 92, 105, 347
őrfeltétel 335, 341, 350
őstípus 159

P

`package` kulcsszó → csomag
`Page` → `Text_IO`
`Page_Length` → `Text_IO`
paraméter 38, 102, 104, 309, 311, 315, 317
aktuális 42, 102
alapértelmezett értéke → feltételezett értéke
alprogram paraméter 102
feltételezett értéke 112, 229, 235, 249
formális 42, 102
módja 105, 234
sablon paraméter *234*
paraméterátadás 151, 158
cím szerinti 151, *524*
eredmény szerinti *524*
érték szerinti *524*
név szerinti *524*
paraméterlista 104
partíció 171, 485
azonosító (`Partition_ID`) 486
hívó 490
hívott 490

konfigurálása 486
passzív szerkezet 334
példány *227*, 251
példányosítás 145, 146, 160, *227*, 231
`Point` → `SVGALib`
polárkoordináták 507
polimorfizmus 261
tágabban értelmezett 105, 113, 227
`Positive` típus 40
potenciálisan blokkoló művelet 342
pozicionális forma *112*, 231
pozíciószám 46, 47, 50
pragma 45
`All_Calls_Remote` 490
`Asynchronouos` 491
`Atomic` 348, 372, 469
`Atomic_Components` 348, 372, 469
`Attach_Handler` 466
`Controlled` 372
`Convention` 348, 373, 444, 455, 461
`Discard_Names` 468
`Elaborate` 373
`Elaborate_All` 373
`Elaborate_Body` 144, 373
`Export` 373, 443, 455, 461
`Import` 116, 144, 348, 374, 443, 455, 461
`Inline` 374
`Interrupt_Handler` 349, 466
`Interrupt_Priority` 473
kategorizációs 486
`Linker_Options` 374, 444
`List` 374
`Locking_Policy` 477
`Optimize` 374
`Pack` 356
`Page` 375
`Preelaborate` 375
`Priority` 473
`Queuing_Policy` 478
`Remote_Call_Interface` 489
`Remote_Types` 488
reprezentációs 355
`Restrictions` 375
`Shared_Passive` 487
`Storage_Size` 358
`Suppress` 140, 375
`Task_Dispatching_Policy` 476
`Volatile` 348, 469
`Volatile_Components` 348,469
precedencia 52, 53, 59, 60
primitív
alprogram 38, 41
eljárás 152
művelet 38, 42, 263, 266
operátor 168
`Priority_Queuing` 478
private kulcsszó 165, 235, 339
prímszám 54
procedure kulcsszó *102*, 335
profil 165
programegység 27, 91, 143, 144, 171, 347,. *521*
specifikációs rész *521*
törzs 91, *521*
deklarációs rész *521*
utasíthatósorozat *521*
`Program_Error` kivétel 126, 346
programkönyvtár 143
programozási nyelv
erősen típusos *523*
szemantika *520*
szintaxis *520*
programszöveg 163
protected hívási konvenció 348
protected kulcsszó 334
`Put` → `Text_IO`
`Put_Line` → `Text_IO`

R

`Raise_Exception` eljárás 133, 257
`raise` utasítás *131*
randevú 309, *316*
`Random` → numerikus csomag
`range` intervallum 38
`range` kulcsszó 243
`Read` → `Sequential_IO`,
→ `Direct_IO`,
→ `Stream_IO`
`record` kulcsszó → rekord
reentráns 103
referencia 267
referenciaképzés 368
rekord 37, 241, 311, 360
- aggregátum *70*
- diszkrimináns *71*, 73, 76
- jelölt 262, *262*
 - diszkrimináns 266
- kiterjesztés 41, 152
- korlátozott *69*
- **`limited`** *69*
- **`tagged`** 263
- típus 68
 - művelet 68
- variáns 73, 268

rekurzió 113, 230
rekurzív típus 255
relációs operátor 41, 45, 55
`rem` operátor 55
`renames` utasítás → átnevezés
`Replace_Element` → sztringkezelés
`Replace_Slice` → sztringkezelés
`Replicate` → sztringkezelés
reprezentációs attribútum 356
reprezentációs elem 355
reprezentációs pragma 355
`requeue` utasítás 343, 351
`Reraise_Occurence` eljárás 134
`Reset` → `Sequential_IO`,
→ `Direct_IO`,
→ `Text_IO`,
→ `Stream_IO`
`return` utasítás *92*, 102, 103
`reverse` kulcsszó → ciklus
részegység 160
részhalmaz 38
résztömb 42, 92
`RGBColour` → `SVGALib`
`root_integer` 42, 379
`root_real` 42, 380
`Root_Storage_Pool` típus 369
rövidzár operátor *52*

S

sablon 138, *227*
- alprogram
 - → alprogram sablon
- csomag 159, 173, 228, 251
 - üres → absztrakció
- sablon egység 165
- eljárás → eljárás sablon
- függvény → függvény sablon
- paraméter
 - absztrakt típus *240*
 - **`aliased`** komponensű tömb *246*
 - alprogram *247*
 - alprogramra mutató típus *246*
 - attribútum aktuálisként *248*
 - átlátszatlan típus *235*
 - csomag 250
 - decimális típus *244*
 - diszkrét típus *243*
 - diszkriminánssal rendelkező típus *237*
 - előjeles egész típus *243*
 - fixpontos típus *244*
 - indefinit típus *238*
 - jelölt típus *239*
 - kiterjesztett típus *241*
 - korlátozott átlátszatlan típus *236*
 - lebegőpontos típus *244*

moduló típus *243*
mutató típus *246*
objektum *234*
operátor *248*
osztályszintű típus *239*
skalár típus *243*
származtatott típus *240*
tömb típus *244*
`Save` 419
`Save_Occurence` eljárás 134
`Scaling` 422
`Screen_Off` → `SVGALib`
`Screen_On` → `SVGALib`
`select` utasítás 309, 318, 319, 321, 324
`else` ág 321
nyitott ág 320
zárt ág 320
`Sequential_IO` csomag *425*
`Set_Col` 204, → `Text_IO`
`Set_Color` → `SVGALib`
`Set_Colour` → `SVGALib`
`Set_Default_Event_Handler` → `SVGALib`
`Set_Error` → `Text_IO`
`Set_Flip_Char` → `SVGALib`
`Set_Index` → `Direct_IO`, → `Stream_IO`
`Set_Input` → `Text_IO`
`Set_Line` → `Text_IO`
`Set_Line_Length` → `Text_IO`
`Set_Mode` → `Stream_IO` → `SVGAlib`
`Set_Output` → `Text_IO`
`Set_Page_Length` → `Text_IO`
`Set_Palette` → `SVGALib`
`Sin` → numerikus csomag
`Sinh` → numerikus csomag
`Size` → `Direct_IO`, → `Stream_IO`
skalár 92
skalár típus 37, 41, 48, 243
`Skip_Line` → `Text_IO`
`Skip_Page` → `Text_IO`
`Slice` → sztringkezelés
small érték 47, *57*, 58
sorhossz → `Text_IO`
speciális karakter 31
specifikációs rész → programegység
specifikus típus *42*
`Sqrt` → numerikus csomag
`Standard` csomag 160, 164, 173, *377*
`Standard_Error` → `Text_IO`
`Standard_Input` → `Text_IO`
`Standard_Output` → `Text_IO`
statikus változó 144
`Status_Error` 221
`Storage_Error` kivétel 77, 126
`Storage_IO` *196*, 425, *428*
`Storage_Unit` konstans 365
`Stream` függvény 218, 220
`Streams.Stream_IO` csomag *217*, *437*
`Strings.Bounded` csomag 397
`String` típus 67
`SVGALib` csomag *223*
`svgalib.ads` *223*
`svgalibmouse.ads` *225*
`Synchronous_Task_Control` csomag 482
`System` csomag 377

Sz

szám 32, 44, 47, 55–58
számjegy 31, 47, 57, 58
számrendszer 32, 57
származtatás 40
közvetett 41, 42
közvetlen 41, 42
származtatási fa 42
származtatási osztály *41*
származtatott típus 38, *40*, 41, 240
szekvenciális fájl → fájl
szelekciós jelölés 97

szelektív várakoztatás 319, 322
szelektor 163
szemafor 340, 349
szemantika 520
szemantikusan függő egység 165
szemantikus környezet 168
szigorú numerikus mód 505, 516
szimbólum 33, 35
szinkronizációs szerkezet 334, 350
szintaktikai egység 33
szintaxis 520
szóköz 31
szövegfájl → fájl
sztringkezelés *386*
 fix sztring *390*
 halmaz *413*
 korlátos sztring *397*
 leképezés *413*
 nemkorlátos *406*
 `Wide_String` *414*
sztringliterál 32
szülő típus 41

T

`tagged` kulcsszó 239, 263
`Tags.Expanded_Name` 469
`Tail` → sztringkezelés
`Tan` → numerikus csomag
`Tanh` → numerikus csomag
`Task_Attributes` csomag 471
`Task_ID` 470
`Task_Identification` csomag 470
`Tasking_Error` kivétel 126, 140, 313, 324
taszk 37, 42, 151, 159, 165, *309*
 aktív prioritás 473
 alapprioritás 473
 attribútum 471
 állapot 475
 belépési pont 333
 besorolási technika 478
 előzés 475
 indítás 475
 kész-sor *475*
 kivételkezelő része *127*
 kizárási technika 477
 `Ceiling_Locking` 477
 komplett 313, 315, 324
 objektum *312*
 prioritás 473
 `Any_Priority` altípus 474
 dinamikus prioritáskezelés 479
 `Interrupt_Priority` altípus 474
 öröklődés 473, 474
 `Priority` altípus 474
 `System.Priority` altípus 473
 szinkronizálás 309
 ütemezési pont 475, 476
 ütemező technika 476
 `FIFO_Within_Priorities` 476
 vezérlés
 aszinkron 483
 szinkron 482
tárfelszabadítás 371
tárgyprogram *521*
tároló egység 485
távoli mutató típus 488
teljes kompatibilitás 104, 107, 116
teljes nézet 160
`terminate` utasítás 309, 318, *320*
terminálás 159
textfájl → fájl
`Text_IO` csomag 425, *429*
`Text_IO.Editing` csomag 498, 501
`Text_IO.Enumeration_IO` csomag 469
`Text_IO.Text_Streams` *439*
`then` kulcsszó → elágazás
`Tick` konstans 365
tizedespont 32
típus *35*, 36, 77, 91, 168

absztrakt 158, 282
adatfolyam 303
attribútum 153
átlátszatlan 158, 264
 korlátozott *150*, 236
beépített 36
deklaráció *36*, 42
diszkrét 37, 48
diszkrimináns 159
egész 44, 48, 53, 55
ekvivalens 36
elemi 37, 41, 159
ellenőrzött *158*
 `Adjust` eljárás 158
 `Controlled` típus 158
 `Finalize` eljárás 158
 `Initialize` eljárás 158
 `Limited_Controlled` típus 158
érték 35, 36, 41, 42
értékhalmaz 38, 40, 42
felsorolási 35, 37, 48–50, 164
fixpontos 37, 57
gyökér 41, 42
jelölt 42
karakter 51
kiterjesztett 146
kiterjesztés
 átlátszatlan 146, *152*
konténer 289
konverzió 153, 366
korlátozott 41, 91, 150, 158, 347
lebegőpontos 37, 56
leszármazott 153
`Lista` 289
logikai 41, 51
megszorított 148, 153
moduló 53
mutató 77
művelet 35, 38, 40, 41, 230
név 36, 38, 41, 42
névtelen 80
nézet
 illeszkedési szabályok *152*
 jelölt 153
 nem jelölt 153
 részleges 152
 teljes 152
objektum 42
operátor 38
osztály *37*, 41, 146, 230
osztályszintű 41, 274
osztályszintű altípus 153
ős 152
ős altípus 153
összetett 37
paraméter 146
rekord → rekord
sablon paramétere 228
skalár 37, 41, 48
specifikáció 261
statikus egyezés 153
származtatási osztály 41
származtatott 38, 41, 73
szülő 41
taszk
 névtelen 310
tömb → tömb
univerzális 41, 42
valós 37, 44, 56
védett 37
`To_Basic` 384, 385
`To_Bounded_String` → sztringkezelés
`To_Character` 386
`To_Domain` → sztringkezelés
`To_ISO_646` 385
tokbazárás 143
`To_Lower` 384, 385
`To_Mapping` → sztringkezelés
`To_Range` → sztringkezelés
`To_Sequence` → sztringkezelés
`To_Set` → sztringkezelés
`To_String` → sztringkezelés
`To_Unbounded_String` → sztringkezelés
`To_Upper` 384, 385

- To_Wide_Character 386
- To_Wide_String 386
- többszörös deklaráció 43
- tömb 37, 40, 41, 92, 147, 311, 318
 - aggregátum 66
 - aliased komponensű 246
 - attribútum 64
 - 'First 65
 - 'First(N) 65
 - 'Last 65
 - 'Last(N) 65
 - 'Length 65
 - 'Length(N) 65
 - 'Range 65
 - 'Range(N) 65
 - elem 42
 - elem típus 61
 - index 61
 - korlátozott 61
 - megszorítás nélküli indexhatárú 238
 - művelet 64
 - típus 61
 - megszorítás nélküli 62
 - megszorított 62
- törzs → programegység
- Translate 393, 403
- Trim → sztringkezelés
- Trim_End → sztringkezelés
- Truncation → sztringkezelés
- type kulcsszó 36, 338

U

- újrafelhasználhatóság *18*, 259
- újrakiválasztás 282
- Unchecked_Conversion 366
- Unchecked_Deallocation 158, 292, 371
- unió típus 73
- universal_fixed 42, 380
- universal_integer 42, 54, 65
- universal_real 42
- univerzális számkonstans *44*
- univerzális típus 41, 42, *42*
- Use_Error kivétel 221
- use utasítás 143, 166, *168*, 229, 230, 232, 251, 252
- utasítás 40, 43–45, 51, *91*, 102
 - egyszerű *91*
 - összetett *93*
- utasítás-orientált nyelv 520
- utasítássorozat 91
 - inicializáló 144

V

- valós idejű rendszerek 349
- valós
 - osztály 42
 - szám 56, 57
 - típus 37, 44, *56*
- Value → sztringkezelés
- változó 36–38, 40, 42, 43, 49, 53, 77–79, 82, 84, 85, 91
 - deklaráció 127
- várakozási sor → belépési pont várakozási sora
- vezérlési szerkezet 91
- védett
 - alprogram *334*
 - egység *338*
 - eljárás 146, *334*
 - függvény *334*
 - objektum 42, 334, *338*
 - belépési pontja *334*
 - belépési pontjához rendelt törzs 335
 - belépési pontjának őrfeltétele 335, 341
 - egyedüli *338*
 - típus 37, 165, *338*
 - diszkriminánsa 340, 351
- végrehajtási idő *521*
- véletlenszám → numerikus csomag
- VgaMode → SVGALib
- visszatérési érték 102, 105

W

`when` kulcsszó 128, 335
`while` utasítás → ciklus
`Wide_Character` típus 51, 381
`Wide_String` típus 67
`Wide_Text_IO` csomag 425
`Wide_Text_IO.Editing` csomag 498, 504
`Wide_Text_IO.Text_Streams` csomag *439*
`with` kulcsszó 228, 247, 251
`with` záradék 143, 161, 166, 230
`Word_Size` konstans 365
`Write` → `Sequential_IO`, → `Direct_IO`

X

`xor` operátor 52

Z

zárójel 104

„ A tankönyv egységes, összefoglaló képet ad az Ada95 programozási nyelvvel kapcsolatos eredményekről. Alapvető tulajdonságainak, lehetőségeinek bemutatásán túl (a nyelv kialakításának, céljainak megfelelően) általános programozási módszertant, stílust is „ tanít”. Az Ada nyelv megismerése segített hozzá, hogy igazából egyetlen nyelv sem tudott „ meglepetést” okozni, legfeljebb egy kicsit más formában kell a programot megírni.”

Kiss Viktória

2850 Ft.

ISBN 963 463 238 6

9 789634 632382